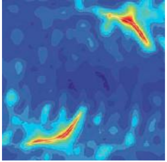


Une école
en mouvement

MO101: Python and Shell Script

Vladimir Paun
ENSTA ParisTech

Shell Script

*A **shell script** is a computer program designed to be run by the Unix shell, a command-line interpreter. The various dialects of shell scripts are considered to be **scripting languages**.*

Shell Script

Typical Unix/Linux/Posix-compliant installations include:

- Korn Shell (ksh)
- Bourne shell (sh) - one of the oldest still in use
- C Shell (csh),
- Bourne Again Shell (**bash**),
- a remote shell (rsh),
- a secure shell for SSL telnet connections (ssh), etc.

Shell Script

Typical Unix/Linux/Posix-compliant installations include:

- Korn Shell (ksh)
- Bourne shell (sh) - one of the oldest still in use
- C Shell (csh),
- Bourne Again Shell (**bash**),
- a remote shell (rsh),
- a secure shell for SSL telnet connections (ssh), etc.

Other shells available based on programmes such as

- **Python**,
- Ruby,
- C,
- Java,
- Perl, etc.

Verisimilitude

The **invocation** of *shell scripts* interpreters is handled as a **core operating system feature**.

Verisimilitude

The **invocation** of *shell scripts* interpreters is handled as a **core operating system feature**.

Shell scripts are set up and executed by the ***OS*** *itself*

Verisimilitude

The **invocation** of *shell scripts* interpreters is handled as a **core operating system feature**.

*Shell scripts are set up and executed by the **OS** itself*

Modern shell script:

- not just on the same footing as system commands
- many system commands are actually shell scripts

  *or just scripts as some of them are not interpreted by a shell, but instead by Perl, **Python**, or some other language*

Shell Script - Life Cycle

Shell scripts often:

- serve as an initial stage in software development, and
- later serve to a different underlying implementation, most commonly being converted to Perl, Python, or C.

While files with the ".sh" file extension are usually a shell script of some kind, most shell scripts do not have any filename extension.

Shell Script - Life Cycle

Shell scripts often:

- serve as an initial stage in software development, and
- later serve to a different underlying implementation, most commonly being converted to Perl, Python, or C.

The interpreter directive:

- *implementation detail* - **fully hidden** inside the script,
- no exposed *filename extension*,
- provides for *seamless reimplementation* in different languages with no impact on end users.

While files with the ".sh" file extension are usually a shell script of some kind, most shell scripts do not have any filename extension.

Advantages and disadvantages

+ *commands* and *syntax* are exactly **the same** as those directly entered at the command-line;

Advantages and disadvantages

- + *commands* and *syntax* are exactly **the same** as those directly entered at the command-line;
- + much *quicker to write* than equivalent code in other languages;
  - easy program or file selection
  - quick start
  - interactive debugging

Advantages and disadvantages

- + *commands* and *syntax* are exactly **the same** as those directly entered at the command-line;
- + much *quicker to write* than equivalent code in other languages;
  - easy program or file selection
  - quick start
  - interactive debugging
- - prone to costly errors;

Advantages and disadvantages

- \+ *commands* and *syntax* are exactly **the same** as those directly entered at the command-line;
- \+ much *quicker to write* than equivalent code in other languages;
  - easy program or file selection
  - quick start
  - interactive debugging
- \- prone to costly errors;

  catastrophic (typing?) error: **rm -rf *␣/** instead of **rm -rf */**

Advantages and disadvantages

- \+ *commands* and *syntax* are exactly **the same** as those directly entered at the command-line;
- \+ much *quicker to write* than equivalent code in other languages;
  - easy program or file selection
  - quick start
  - interactive debugging
- \- prone to costly errors;

  catastrophic (typing?) error: **rm -rf *␣/** instead of **rm -rf */**
  or is it the other way around ?

Advantages and disadvantages

+ *commands* and *syntax* are exactly **the same** as those directly entered at the command-line;
+ much *quicker to write* than equivalent code in other languages;
  - easy program or file selection
  - quick start
  - interactive debugging
- prone to costly errors;

  catastrophic (typing?) error: **rm -rf *␣/** instead of **rm -rf */**
  or is it the other way around ? **NO!**

Advantages and disadvantages

- + *commands* and *syntax* are exactly **the same** as those directly entered at the command-line;
- + much *quicker to write* than equivalent code in other languages;
  - easy program or file selection
  - quick start
  - interactive debugging
- - prone to costly errors;

  catastrophic (typing?) error: **rm -rf *␣/** instead of **rm -rf */**
  or is it the other way around ? **NO!**
- - slow *execution speed*
- - need to launch a new process for most of executed command
- - pipelining helps, but complex script are several orders of magnitude slower than equivalent compiled program

Shell Script - typical Linux flow

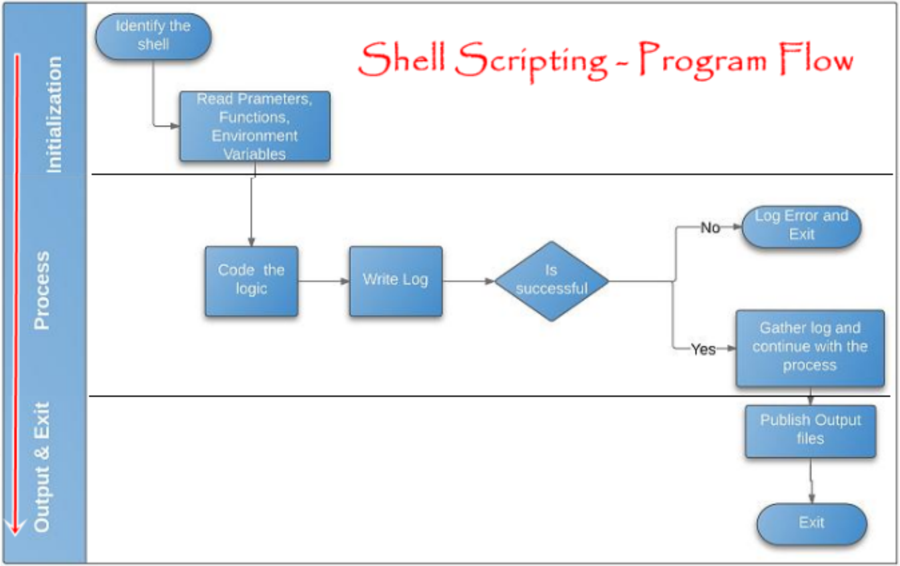
```
Input Disk ----> sh/ksh/bash ------------------> Output Disk
                   |                       ^
                   v                       |
                    --> Python script ------
                   |                       ^
                   v                       |
                    --> awk script ---------
                   |                       ^
                   v                       |
                    --> sed script ----------
                   |                       ^
                   v                       |
                    --> C/C++ program ------
                   |                       ^
                   v                       |
                    --- Java program -------
                   |                       ^
                   v                       |
                   :                       :
```

Shells are the glue of Linux

Linux shells like sh/ksh/bash/...

- provide input/output/flow-control designation facilities
- they are Turing complete languages in their own right while
- optimized to efficiently pass data and control to and from other executing processes written in any language the O/S supports.

Shell Scripting - Program Flow



Scripting vs. Compiled Languages

Scripting languages

- Bash, Python, Perl, postscript, matlab/octave, ...
- Interactive mode
- Few optimisations
- Easy to use
- No binary files (hardly useful for commercial software)

Scripting vs. Compiled Languages

Scripting languages

- Bash, Python, Perl, postscript, matlab/octave, ...
- Interactive mode
- Few optimisations
- Easy to use
- No binary files (hardly useful for commercial software)

Compiled languages

- C, C++, Fortran, ...
- Efficient for computational expensive tasks
- Source code is compiled to binary code

What to choose?

Unix Shell vs. Python

Unix Shell vs. Python

Shell (Bash)

- Seperate program for each simple task
- Gluing together programs with a script
- Not really a full programming language
- Powerful tools available
- Suitable for small tools (1-100 lines of code)

Unix Shell vs. Python

Shell (Bash)

- Seperate program for each simple task
- Gluing together programs with a script
- Not really a full programming language
- Powerful tools available
- Suitable for small tools (1-100 lines of code)

Python

- Full programming language (one that you **know already**)
- Large number of libraries available
- Intuitive naming conventions
- Suitable for almost any task

Use of Shell Scripts

Where **to use** them:

- System administration
- Automating everyday terminal tasks
- Searching in and manipulating ASCII-Files, etc.

Use of Shell Scripts

Where **to use** them:

- System administration
- Automating everyday terminal tasks
- Searching in and manipulating ASCII-Files, etc.

Where **not to use** them:

- Lots of mathematical operations
- Computational expensive tasks
- Large programs which need structuring and modularisation
- Arbitrary file access
- Data structures
- Platform-independent programs, etc.

General Programming Rules

- Comments
- Comments
- Comments
- Problem $\Rightarrow$ algorithm $\Rightarrow$ program
- Modular programming
- Tests
- Generic where possible, specific where necessary
- ...

Writing Python Scripts

Python

- great flexible programming language
  - imperative
  - interpreted
  - object oriented
- can be used in many situations.

Writing Python Scripts

Python

- great flexible programming language
  - imperative
  - interpreted
  - object oriented
- can be used in many situations.

In this lecture we use Python to enhance the Unix/Linux shell environment.

Writing Python Scripts

Python

- great flexible programming language
  - imperative
  - interpreted
  - object oriented
- can be used in many situations.

In this lecture we use Python to enhance the Unix/Linux shell environment.

Practical Note: You can even name your shell scripts with the **.sh** extension and run them as you would run any bash shell script.

Writing Python Scripts - Example

Python scripts should start with the following line of code:

```
#! /usr/bin/env python
for i in range(4):
          print(i,)
```

Writing Python Scripts - Example

Python scripts should start with the following line of code:

```
#! /usr/bin/env python
for i in range(4):
          print(i,)
```

Of course this depends on where you have python installed.

Writing Python Scripts - Example

Python scripts should start with the following line of code:

```
#! /usr/bin/env python
for i in range(4):
          print(i,)
```

Of course this depends on where you have python installed.

Similar with bash scripts, remember?

Writing Python Scripts - Example

Python scripts should start with the following line of code:

```
#! /usr/bin/env python
for i in range(4):
          print(i,)
```

Of course this depends on where you have python installed.

Similar with bash scripts, remember?

Don’t forget to make the file executable. How?

Writing Python Scripts - Example

Python scripts should start with the following line of code:

```
#! /usr/bin/env python
for i in range(4):
          print(i,)
```

Of course this depends on where you have python installed.

Similar with bash scripts, remember?

Don't forget to make the file executable. How?

$> chmod +x scriptName.py

Writing Python Scripts - Example

Python scripts should start with the following line of code:

```
#! /usr/bin/env python
for i in range(4):
        print(i,)
```

Of course this depends on where you have python installed.

Similar with bash scripts, remember?

Don't forget to make the file executable. How?

$> chmod +x scriptName.py

and put it in a directory on your PATH (can be a symlink):

cd /bin/
ln -s /some/path/to/myscript/scriptName.py

Writing Python Scripts - Example

A Python script:

```
#! /usr/bin/env python
for i in range(4):
          print(i,)
```

Writing Python Scripts - Example

A Python script:

```
#! /usr/bin/env python
for i in range(4):
          print(i,)
```

Now you can execute your script by using the following command:

Writing Python Scripts - Example

A Python script:

```
#! /usr/bin/env python
for i in range(4):
          print(i,)
```

Now you can execute your script by using the following command:

$> ./scriptName.py

Remember that you can loose the .py extension.

The `sys` module

If we import `sys`, then the command line content is stored in the `sys.argv` list. For example:

```
#! /usr/bin/python
# getlist.py
import sys
print (sys.argv,)
```

Then if we type

```
./getlist.py file1 file2 file3
```

the script would print

```
(['./getlist.py', '1', '2', '3'],)
```

Note: `sys.argv` contains the name of the file.

sys - Years till 100

Write a python script that takes 2 command line arguments:

- name of a person
- age of the person

and checks how many years that person has left until reaching the age 100.

The script will be called using:

```
./years.py Joe 25
```

sys - Years till 100

```
#!/usr/bin/env python
import sys
if len(sys.argv) > 1:
    name = sys.argv[1]
else:
    name = raw_input('Enter Name:')
if len(sys.argv) > 2:
    age = int(sys.argv[2])
else:
    age = int(raw_input('Enter Age:'))
sayHello = 'Hello ' + name + ','
if age == 100:
    sayAge = 'You are already 100 years old!'
elif age < 100:
    sayAge = 'You will be 100 in ' + str(100 - age) + ' years!'
else:
    sayAge = 'You turned 100 ' + str(age - 100) + ' years ago!'
print(sayHello, sayAge)
```

use:

```
./years.py Joe 25
```

optparse

The python class **optparse.OptionParser** - a powerful tool for creating options for your script.

Previous example:

- ok we had the user enter two command line arguments to the python script,
- ko no specification of which is which.

optparse

The python class **optparse.OptionParser** - a powerful tool for creating options for your script.

Previous example:

- ok we had the user enter two command line arguments to the python script,
- ko no specification of which is which.

Better: be able to give parameters in any specific order and specify which is which

We can do this in python very easily, using the OptionParser module.

`optparse`

The `OptionParse` class

- add options to your script
- generate a help option based on the options you provide.

optparse

The `OptionParse` class

- add options to your script
- generate a help option based on the options you provide.

We are adding two options:

- **-n** (or **–name**)
- **-a** (or **–age**).

optparse

The parameters of `add_option`

1. the short option and
2. the long option (it is common in the Unix to add a short and long version of an option)
3. **dest=**, the variable name created,
4. **help=**, the help text generated and
5. **type=**, the type for the variable. By default the type is string, but for age, we want to make it int.

optparse

The parameters of `add_option`

1. the short option and
2. the long option (it is common in the Unix to add a short and long version of an option)
3. **dest=**, the variable name created,
4. **help=**, the help text generated and
5. **type=**, the type for the variable. By default the type is string, but for age, we want to make it int.

```
#!/usr/bin/env python
import sys, optparse

parser = optparse.OptionParser()

parser.add_option('-n', '--name', dest='name', help='Your Name')
parser.add_option('-a', '--age', dest='age', help='Your Age', type=int)
```

optparse

After adding the options, we call the parse_args function, which will return:

- an options object,
- an args list object.

We can now access the variables defined in "dest=" on the options object returned.

```
(options, args) = parser.parse_args()
if options.name is None:
    options.name = raw_input('Enter Name:')
if options.age is None:
    options.age = int(raw_input('Enter Age:'))
sayHello = 'Hello ' + options.name + ','
if options.age == 100:
    sayAge = 'You are already 100 years old!'
elif options.age < 100:
    sayAge = 'You will be 100 in ' + str(100 - options.age) + ' years!'
else:
    sayAge = 'You turned 100 ' + str(options.age - 100) + ' years ago!'
```

Will have two options, options.name and options.age.
Also checks if one of the variables wasn't passed.

optparse - run the example

./years.py	Prompts for user and age
./years.py -n Joe	Sets user, prompts for age
./years.py --name Joe	Sets user, prompts for age
./years.py -a 25	Sets age, prompts for user
./years.py --age 25	Sets age, prompts for user
./years.py -a 25 --name Joe	Sets age, sets user
./years.py -n Joe --age 25	Sets age, sets user

Another thing you can do now is run the help option, by specifying either `-h` or `- -help`:

```
./years.py -h
#This will give the following output and then exit the script:
usage: years.py [options]

options:
  -h, --help            show this help message and exit
  -n NAME, --name=NAME  Your Name
  -a AGE, --age=AGE     Your Age
```

optparse

```
#!/usr/bin/env python
import sys, optparse

parser = optparse.OptionParser()
parser.add_option('-n', '--name', dest='name', help='Your Name')
parser.add_option('-a', '--age', dest='age', help='Your Age', type=int)
(options, args) = parser.parse_args()
if options.name is None:
    options.name = raw_input('Enter Name:')
if options.age is None:
    options.age = int(raw_input('Enter Age:'))
sayHello = 'Hello ' + options.name + ','
if options.age == 100:
    sayAge = 'You are already 100 years old!'
elif options.age < 100:
    sayAge = 'You will be 100 in ' + str(100 - options.age) + ' years!'
else:
    sayAge = 'You turned 100 ' + str(options.age - 100) + ' years ago!'
print(sayHello, sayAge)
```

The subprocess module

One of the most useful packages for unix shell scripters in python is the **subprocess** package.

The `subprocess` module

One of the most useful packages for unix shell scripters in python is the **subprocess** package.

The simplest use of this package is to use the call function to call a shell command:

The `subprocess` module

One of the most useful packages for unix shell scripters in python is the **subprocess** package.

The simplest use of this package is to use the call function to call a shell command:

```
#!/usr/bin/env python
import subprocess
subprocess.call("ls -l", shell=True)
```

The `subprocess` module

One of the most useful packages for unix shell scripters in python is the **subprocess** package.

The simplest use of this package is to use the call function to call a shell command:

```
#!/usr/bin/env python
import subprocess
subprocess.call("ls -l", shell=True)
```

This script will call the unix command "`ls -l`" and print the output to the console.

The `subprocess` module

One of the most useful packages for unix shell scripters in python is the **subprocess** package.

The simplest use of this package is to use the call function to call a shell command:

```
#!/usr/bin/env python
import subprocess
subprocess.call("ls -l", shell=True)
```

This script will call the unix command "`ls -l`" and print the output to the console.

Useful, however you might want to **process the results** of the call inside your script instead of just **printing them to the console**.

subprocess

To do this, you will need to open the process with the `Popen` function: takes an array containing

- the process to invoke
- its command line parameters.

So if we wanted to tail the last 500 lines of a log file, we would pass in each of the parameters as a new element in the array. The following script shows how:

subprocess

To do this, you will need to open the process with the `Popen` function: takes an array containing

- the process to invoke
- its command line parameters.

So if we wanted to tail the last 500 lines of a log file, we would pass in each of the parameters as a new element in the array. The following script shows how:

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

proc = subprocess.Popen(['tail', '-500', 'mylogfile.log'],
 stdout=subprocess.PIPE)

for line in proc.stdout.readlines():
    print(line.rstrip())
```

It pipes the output back to your script via the "proc.stdout" variable + loop. This script will open the process on unix "`tail -500 mylogfile.log`", read the output of the command and print it to the console.

os module

The os module contains lots of useful things as well

- `os.path.exists('path')` - test if a path exists
- `os.path.isfile('file')` - test if its a file
- `os.path.isdir('dir')` - test if its a folder

and lots of other things including changing directories, deleting files and changing permissions.

Compressing files

Example: module `gzip`

- Simple to use: just replace standard call to `open`

```
import gzip
fd = gzip.open("file.txt.gz", "w")
fd.write("""Funny lines in gzip file""")
fd.close()
fd = gzip.open("file.txt.gz")

print(fd.read())
```

...as easy as that File modes as usual (rwa + b for binary)

A simple HTTP server

Web server in 3 lines?

```
import SimpleHTTPServer, SocketServer
httpd = SocketServer.TCPServer(("", 8000), \
    SimpleHTTPServer.SimpleHTTPRequestHandler)
httpd.serve_forever()
```

- What could this be good for?
  - Want to share quickly some files with colleagues in the same network?
  - ⇒ Goto directory, start python, run three lines, tell them your IP and the port (here: 8000)
  - That'sit!

Web server in 3 lines?

```
import SimpleHTTPServer, SocketServer
httpd = SocketServer.TCPServer(("", 8000), \
    SimpleHTTPServer.SimpleHTTPRequestHandler)
httpd.serve_forever()
```

- What could this be good for?
  - Want to share quickly some files with colleagues in the same network?
  - ⇒ Goto directory, start python, run three lines, tell them your IP and the port (here: 8000)
  - That'sit!

Regular Expressions - Introduction

The `re` module was added in Python 1.5, and provides Perl-style regular expression patterns.

Regular Expressions - Introduction

The `re` module was added in Python 1.5, and provides Perl-style regular expression patterns.

What?

Regular expressions (called REs, or regexes, or regex patterns) are essentially a ***tiny****,* ***highly specialized*** *programming language embedded inside Python and made available through the re module.*

Regular Expressions - Introduction

The `re` module was added in Python 1.5, and provides Perl-style regular expression patterns.

What?
*Regular expressions (called REs, or regexes, or regex patterns) are essentially a **tiny**, **highly specialized** programming language embedded inside Python and made available through the re module.*

Why?
Using this little language, you specify the **rules** for the **set of possible strings** that you **want to match**.

Regular Expressions - Introduction

The `re` module was added in Python 1.5, and provides Perl-style regular expression patterns.

What?
Regular expressions (called REs, or regexes, or regex patterns) are essentially a ***tiny***, ***highly specialized*** *programming language embedded inside Python and made available through the re module.*

Why?
Using this little language, you specify the **rules** for the **set of possible strings** that you **want to match**.

How?
Regular expression patterns are **compiled** into a series of bytecodes which are then **executed** by a **matching engine written in C**.

Regular Expressions - Use Cases

The set of possible strings you want to match might contain:

- English sentences,

Regular Expressions - Use Cases

The set of possible strings you want to match might contain:

- English sentences,
- e-mail addresses,

Regular Expressions - Use Cases

The set of possible strings you want to match might contain:

- English sentences,
- e-mail addresses,
- TeX commands,

Regular Expressions - Use Cases

The set of possible strings you want to match might contain:

- English sentences,
- e-mail addresses,
- TeX commands,
- ...or anything you like.

Regular Expressions - Use Cases

The set of possible strings you want to match might contain:

- English sentences,
- e-mail addresses,
- TeX commands,
- ...or anything you like.

You can then ask questions such as

- “Does this string match the pattern?”,
- “Is there a match for the pattern anywhere in this string?”,

You can also use REs to modify a string or to split it apart in various ways.

Regular Expressions - Matching Characters

Here's a complete list of the metacharacters;

```
. ^ $ * + ? { } [ ] \ | ( )
```

`[` and `]` used for specifying a set of characters that you wish to match:

- listed individually
- range of characters can be indicated by giving two characters and separating them by a '-'

Regular Expressions - Matching Characters

Here's a complete list of the metacharacters;

```
. ^ $ * + ? { } [ ] \ | ( )
```

`[` and `]` used for specifying a set of characters that you wish to match:

- listed individually
- range of characters can be indicated by giving two characters and separating them by a '-'

Example:

- `[abc]` will match any of the characters a, b, or c; (same as `[a-c]`)
- `[^5]` will match any character except '5'.

Regular Expressions - Matching Characters

Here's a complete list of the metacharacters;

```
. ^ $ * + ? { } [ ] \ | ( )
```

`[` and `]` used for specifying a set of characters that you wish to match:

- listed individually
- range of characters can be indicated by giving two characters and separating them by a '`-`'

Example:

- `[abc]` will match any of the characters a, b, or c; (same as `[a-c]`)
- `[^5]` will match any character except '5'.

Question: How to match only lowercase letters?

Regular Expressions - Matching Characters

Here's a complete list of the metacharacters;

```
. ^ $ * + ? { } [ ] \ | ( )
```

[and] used for specifying a set of characters that you wish to match:

- listed individually
- range of characters can be indicated by giving two characters and separating them by a '-'

Example:

- [abc] will match any of the characters a, b, or c; (same as [a-c])
- [^5] will match any character except '5'.

Question: How to match only lowercase letters?
Answer: your RE would be [a-z]

Regular Expressions - the backslash \

Some of the special sequences beginning with 'ŕepresent predefined sets of characters that are often useful

- **\d** : decimal digit; this is equivalent to the class **[0-9]**.
- **\D** : non-digit character; this is equivalent to the class **[^0-9]**.
- **\s** : whitespace character; this is equivalent to the class **[\t\n\r\f\v]**.
- **\S** : non-whitespace character; this is equivalent to the class **[^\t\n\r\f\v]**.
- **\w** : alphanumeric character; this is equivalent to the class **[a-zA-Z0-9_]**.
- **\W** : non-alphanumeric character; this is equivalent to the class **[^a-zA-Z0-9_]**.

Regular Expressions - Repetition

You can specify that portions of the RE must be repeated a certain number of times.
The first metacharacter for repeating things that we'll look at is *.

- * doesn't match the literal character *;
- specifies that the previous character can be matched zero or more times, instead of exactly once.

Regular Expressions - Repetition

You can specify that portions of the RE must be repeated a certain number of times.
The first metacharacter for repeating things that we'll look at is `*`.

- * doesn't match the literal character `*`;
- specifies that the previous character can be matched zero or more times, instead of exactly once.

Example: `ca*t` **will match**

Regular Expressions - Repetition

You can specify that portions of the RE must be repeated a certain number of times.
The first metacharacter for repeating things that we'll look at is `*`.

- `*` doesn't match the literal character `*`;
- specifies that the previous character can be matched zero or more times, instead of exactly once.

Example: `ca*t` **will match**

- `ct` (0 **a** characters),

Regular Expressions - Repetition

You can specify that portions of the RE must be repeated a certain number of times.
The first metacharacter for repeating things that we'll look at is `*`.

- `*` doesn't match the literal character `*`;
- specifies that the previous character can be matched zero or more times, instead of exactly once.

Example: `ca*t` **will match**

- `ct` (0 **a** characters),
- `cat` (1 **a**),

Regular Expressions - Repetition

You can specify that portions of the RE must be repeated a certain number of times.
The first metacharacter for repeating things that we'll look at is `*`.

- `*` doesn't match the literal character `*`;
- specifies that the previous character can be matched zero or more times, instead of exactly once.

Example: `ca*t` **will match**

- `ct` (0 **a** characters),
- `cat` (1 **a**),
- `caaat` (3 **a** characters),
- and so forth

Regular Expressions - A step-by-step example

Let's consider the expression **a[bcd]*b**. What does it do?

Regular Expressions - A step-by-step example

Let’s consider the expression **a[bcd]*b**. What does it do?

This matches:

- the letter ’**a**’, followed by
- *zero or more* letters from the class **[bcd]**,
- and finally ends with the letter ’**b**’.

Regular Expressions - A step-by-step example

Regular Expressions `a[bcd]*b`
Matching against `abcbd`.

Regular Expressions - A step-by-step example

Regular Expressions `a[bcd]*b`
Matching against `abcbd`.

Step:

Regular Expressions - A step-by-step example

Regular Expressions `a[bcd]*b`
Matching against `abcbd`.

Step:

1. matched: **a** - The a in the RE matches.

Regular Expressions - A step-by-step example

Regular Expressions `a[bcd]*b`
Matching against `abcbd`.

Step:

1. matched: **a** - The a in the RE matches.
2. matched: **abcbd** - The engine matches **[bcd]***, going as far as it can, which is to the end of the string.

Regular Expressions - A step-by-step example

Regular Expressions `a[bcd]*b`
Matching against `abcbd`.

Step:

1. matched: **a** - The a in the RE matches.
2. matched: **abcbd** - The engine matches **[bcd]***, going as far as it can, which is to the end of the string.
3. matched: **Failure** - The engine tries to match **b**, but the current position is at the end of the string, so it fails.

Regular Expressions - A step-by-step example

Regular Expressions `a[bcd]*b`
Matching against `abcbd`.

Step:

1. matched: **a** - The a in the RE matches.
2. matched: **abcbd** - The engine matches **[bcd]***, going as far as it can, which is to the end of the string.
3. matched: **Failure** - The engine tries to match **b**, but the current position is at the end of the string, so it fails.
4. matched: **abcb** - Back up, so that **[bcd]*** matches one less character.

Regular Expressions - A step-by-step example

Regular Expressions `a[bcd]*b`
Matching against `abcbd`.

Step:

1. matched: **a** - The a in the RE matches.
2. matched: **abcbd** - The engine matches **[bcd]***, going as far as it can, which is to the end of the string.
3. matched: **Failure** - The engine tries to match **b**, but the current position is at the end of the string, so it fails.
4. matched: **abcb** - Back up, so that **[bcd]*** matches one less character.
5. matched: **Failure** - Try **b** again, but the current position is at the last character, which is a '**d**'.

Regular Expressions - A step-by-step example

Regular Expressions `a[bcd]*b`
Matching against `abcbd`.

Step:

1. matched: **a** - The a in the RE matches.
2. matched: **abcbd** - The engine matches **[bcd]***, going as far as it can, which is to the end of the string.
3. matched: **Failure** - The engine tries to match **b**, but the current position is at the end of the string, so it fails.
4. matched: **abcb** - Back up, so that **[bcd]*** matches one less character.
5. matched: **Failure** - Try **b** again, but the current position is at the last character, which is a '**d**'.
6. matched: **abc** - Back up again, so that **[bcd]*** is only matching **bc**.

Regular Expressions - A step-by-step example

Regular Expressions `a[bcd]*b`
Matching against `abcbd`.

Step:

1. matched: **a** - The a in the RE matches.
2. matched: **abcbd** - The engine matches **[bcd]***, going as far as it can, which is to the end of the string.
3. matched: **Failure** - The engine tries to match **b**, but the current position is at the end of the string, so it fails.
4. matched: **abcb** - Back up, so that **[bcd]*** matches one less character.
5. matched: **Failure** - Try **b** again, but the current position is at the last character, which is a '**d**'.
6. matched: **abc** - Back up again, so that **[bcd]*** is only matching **bc**.
7. matched: **abcb** - Try **b** again. This time the character at the current position is '**b**', so it succeeds.

RE - Compiling Regular Expressions

Regular expressions are compiled into pattern objects

```
>>> import re
>>> p = re.compile('ab*')
>>> p
<_sre.SRE_Pattern object at 0x...>
```

`re.compile()` also accepts an optional flags argument, used to enable various special features and syntax variations.

```
>>> p = re.compile('ab*', re.IGNORECASE)
```

The RE is passed to `re.compile()` as a string. the `re` module is simply a C extension module included with Python, just like the `socket` or `zlib` modules.

RE - Performing Matches

Regular expressions are compiled into pattern objects

- **match()** Determine if the RE matches at the beginning of the string.
- **search()** Scan through a string, looking for any location where this RE matches.
- **findall()** Find all substrings where the RE matches, and returns them as a list.
- **finditer()** Find all substrings where the RE matches, and returns them as an iterator.

```
>>> import re
>>> p = re.compile('[a-z]+')
>>> p  #doctest: +ELLIPSIS
<_sre.SRE_Pattern object at 0x...>
```

Empty string shouldn't match at all, since + means 'one or more repetitions'.

```
>>> p.match("")
>>> print p.match("")
None
```

Summary

Building with **shell scripts**

- is like assembling a computer with off-the-shelf components the way desktop PCs are.

Building with **Python, C++** or most any other language

- is more like building a computer by soldering the chips (libraries) and other electronic parts together the way smartphones are.