# 

# Introducing Python Script

Mike Jackson, The Software Sustainability Institute



This work is licensed under the Creative Commons Attribution License

Copyright © Software Carpentry and The University of Edinburgh 2010-2012

See http://software-carpentry.org/license.html for more information.

**Things you should do are written in bold.**

Suggested dialog is in normal text.

Command-line excerpts and code fragments are in shaded fixed-width font.

## Introduction

**Start with python-short-intro.ppt slides which cover trade-offs between languages, why we use Python, nature of the course – not to teach programming but to teach good programming hints, tips, guidelines using Python.**

**Try and leave 30m (or more) for the exercise.**

## Basics

Feel free to follow along, typing as I type.

Hint for bash users.

script

Logs the entire set of operating system commands you type and the outputs.

python

Python is interpreted. No separate compilation step.

print "Hello!"

print 1 + 2

print 'Charles' + 'Darwin'

planet = 'Pluto'

No need to declare variables in Python. Created on first use.

print planet

moon = 'Charon'

print moon

print planet

print p

Beware, Python does not assume default variables. Assign variable before use.

p = planet

print p

Variables are just names.

Variables are not typed, unlike Java or C.

planet = 9

print planet

print p

Memory occupied by 'Pluto' is garbage collected or recycled.

string = "two"

number = 3

**Ask students for expected result of the following and add it in as a comment**

# is a comment.

print string \* number # Expect **???**

Repeated concatenation.

**Ask students for expected result of the following**

print string + number

Convert types as values are typed.

print int('2') + 3

print '2' + str('3')

print string + str(number)

Typical arithmetic operations are supported.

print 10 + 3

print 10 / 3

Integer division.

Remainder.

print 10 % 3

Floating point division.

print 10 / 3.0

Complex numbers.

result = print (1 + 4j) + (2 + 5j)

print result = (1 + 4j) + (2 + 5j)

print result.real

print result.imag

String arithmetic.

print 'a' + 'b'

+ is an overloaded operator. How it behaves depends on its arguments.

print 'ab' \* 3

Exponent.

print 2 \*\* 3

print 64 \*\* 0.5

year = 2012

year = year + 1

print year

year += 1

In-place operator.

print year

year /= 2

print year

Typical conditionals are supported.

print 3 < 5

print 3 < 3

Logical values True and False.

print 3 <= 3

print 3 <> 5

print 3 != 5

Both forms of inequality.

print 3 = 5

print 3 == 5

Use double-equals. Legacy of C!

print 3 == 3

Beware, it’s a common error to mistake assignment and equality.

print (3 + 2j) < (5 + 7j)

No such conditional for complex numbers so beware.

## Executing Python files

Interactive interpreter is useful for quick experiments with small code fragments.

Easier to edit a Python file than use the history.

Exit Python.

CTRL-D

Exit script if you started it.

CTRL-D

more typescript

Good for saving interactions with command-line tools or when building complex open source packages.

Always send command-line excerpts in bug reports or e-mails rather than just an English language “interpretation” of the error message!

## Iteration and file reading

In a new terminal window, open a new Python in an editor.

none python-intro.py

**Set python-intro.py contents to be**

print 'Hello'

In a new terminal window, open a file, data.txt, in an editor.

nano data.txt

**Set data.txt contents to be**

Date,Species,Count

2012.04.28,marlin,2

2012.04.28,turtle,1

2012.04.28,shark,3

2012.04.27,marlin,4

**Set python-intro.py contents to be**

source = open('data.txt', 'r')

for line in source:

print line

source.close()

r specifies that file is opened for reading. w is used for writing.

open is a built-in function.

close is a method, on a variable.

Closing the file is good practice. Release when no longer needed. This is important when using databases which may only support a certain number of connections.

source is an iterator. Provides each line in turn.

“read lines as list and loop over list” is a common idiom.

**Run**

python python-intro.py

## Readability

C, Java use {} for indendation. Python uses spaces.

Research shows that people use spaces to determine indentation, so Python's design is better than C or Java, if more problematic for the writer!

4 spaces is recommended.

Never TABs.

Programming is a human activity. Programs are read and manipulated by humans.

Short-term memory can only hold so much at a time (7 +/- 2).

Build things to fit into it.

Indentation.

Method and function names.

Variable names.

**Change "line" to "zonk"**

**Run**

python python-intro.py

**Revert to "line"**

**Run**

python python-intro.py

Balance between "f", "fft" and "fast\_fourier\_transform\_function".

Too short can be too cryptic.

Too long can slow down comprehension.

Typically aim for under 12 characters for names. Most spelling mistakes are at the ends of words.

Computers don't care about these. Code should be readable for humans.

Data should be readable by machines. Can present data in human-readable format.

## Blank lines

Why do we get the blank lines?

End-of-line character in file. print prints on a new line.

String variables support strip method.

**Change "print line" to be**

print line.strip()

**Run**

python python-intro.py

**Add lines to end of python-intro.py**

print dir(line)

print dir(source)

print source.closed

dir is a useful function that shows methods supported by variables.

**Run**

python python-intro.py

**Remove above lines from python-intro.py**

## Correctness and testing

Program runs and works but is it correct?

No as it has not been tested.

If it has not been tested, it is broken!

An "oracle" is a trustworthy provider of results we can test against.

**Run**

wc data.txt

If we trust "wc" we can use it as an oracle.

**Run**

python python-intro.py

Results are the same.

Modify in small testable chunks so you can test as you go, not after half a day of coding without testing.

Easier to fix problems when you're still focused on them than some time later.

Count number of data entries.

**Set python-intro.py contents to be**

source = open('data.txt', 'r')

number = 0

# Count number of data records.

for line in source:

number += 1

source.close()

print "Number of lines:", number

**Run**

python python-intro.py

Skip first line. It is not data.

**Insert before "for line in source"**

source.readline()

**Run**

python python-intro.py

## Comments

Need to comment this. Otherwise how will reader know why we skip the first line.

**Insert before "source.readline()"**

# Read first line.

Poor comment. Code already explains "what".

Comments should explain "why", rationale.

**Change comment to**

# Read title record.

**Change final print to**

print "Number of data records:", number

**Run**

python python-intro.py

Program usefulness directly related to readability.

Readability makes it easier for:

* You to find bugs.
* You understand what you did 6 months from now.
* Your team mates or peers to understand what you did.

Good program is understandable by someone familiar with programming but not with that language.

Code reviews can help you assess this.

Rigorous inspection can remove 60-90% of errors before first test is run (Fagan, 1975).

First review and hour matter most (Cohen, 2006).

Changes in files commited to version control should be small enough to be readable carefully in an hour.

## Conditionals

**Add a comment line to data.txt**

# Data samples observed by John Smith.

Program is now incorrect. Need to skip title and comments.

**Ask students for options to solve this.**

**Remove "source.readline()" and comment.**

**Add conditionals within "for line in source"**

if line.startswith('#'): # Skip comments.

pass

elif line.startswith('D'): # Skip title.

pass

else:

number = number + 1

Remember, comments explain the "why".

pass means "do nothing".

Strings also support endswith. Good design. Exploits expectation.

**Run**

python python-intro.py

Here we now have two great strengths of programs:

* Repetition
* Selection (or conditional execution)

**Ask about problems with assuming "D" for title record.**

Columns might be permuted. Animal might be Dolphin. Count would be wrong.

**Change "# Skip title." to**

# Skip title. FIXME we need a better data format.

# FIXME or # FIXME(1782) can flag work in progress.

Allows you to keep focused on your problem solving.

## Testing conditions

Need to run under various conditions.

**Write 3 data files**

* **comment.txt - with a comment only.**
* **nodata.txt - with a comment, title only.**
* **onedata.txt - with comment, title, one data line.**

How do we test?

**Edit python-intro.py and change 'data.txt' to each file in turn and run**

python python-intro.py

Not very efficient time-wise.

May introduce an error in our program.

Want to automate.

Problem is our hard-coded file name.

## Command-line arguments

Pass in at command-line.

**Create temp.py with following content**

import sys

print sys.platform

print sys.version

print "Arguments are ", sys.argv

print "Number of arguments ", len(sys.argv)

for arg in sys.argv:

print arg

Now we can run

python temp.py

python temp.py data.txt

python temp.py data.txt another.txt

sys.argv is command-line argument values. Copied from C.

**Add line**

print "First entry", sys.argv[0]

First argument is always Python file name.

**Run**

python temp.py data.txt another.txt

**Add following lines to start of python-intro.py**

import sys

if (len(sys.argv) < 2):

sys.exit("Missing file name")

filename = sys.argv[1]

print "File name", filename

**Change "'data.txt'" to be "filename"**

sys.exit(string) exits program.

Assign sys.argv[1] to variable makes role of sys.argv[1] clearer.

Supplement with a comment.

**Add comment**

# A filename is expected as the first command-line argument.

First argument is a file name.

**Run**

python python-intro.py

python python-intro.py data.txt

Now we can run our tests.

**Run**

python python-intro.py comment.txt

python python-intro.py nodata.txt

python python-intro.py onedata.txt

python python-intro.py data.txt

Monotonous.

## Test scripts (pass - covered later?)

Let’s create a script.

**Create python-test.sh with content**

python python-intro.py comment.txt

python python-intro.py nodata.txt

python python-intro.py onedata.txt

python python-intro.py data.txt

Now, if we run it.

**Run**

./python-test.sh

**Edit to add expected values**

echo "Testing...expect 0"

python python-intro.py comment.txt

echo "Testing...expect 0"

python python-intro.py nodata.txt

echo "Testing...expect 1"

python python-intro.py onedata.txt

echo "Testing...expect 4"

python python-intro.py data.txt

**Run**

./python-test.sh

Modify in small testable chunks so you can test as you go, not after half a day of coding without testing.

## Exercise – 30 minutes

We will add a comment

#- N

where N is the number of data records e.g. for data.txt

#- 4

Adding this means the data supports validation.

People are fallible. Make defense a habit. Check your data.

Figure out how to test things before you write them.

Here is a very useful string function.

**Run at the command-line**

python

names = "a b c d"

print names.split()

names = "a,b,c,d"

print names.split(",")

Pair programming.

* Immediate peer review.
* Fewer mistakes.
* More working code per hour.
* Not practical if it's just you.
* Only works for a few weeks as learn to think alike.
* Use sparingly.
* Use for hard problems.
* Use to induct new team members so they get up to speed faster.

In pairs, finish the program off to check that, if the expected number of records is there then, it matches the actual number.

Wave if you get stuck and a helper will come round!

How did you get on?

Sample solution.

# A filename is expected as the first command-line argument.

import sys

if (len(sys.argv) < 2):

sys.exit("Missing file name")

filename = sys.argv[1]

print "File name", filename

source = open(filename, 'r')

number = 0

expected = None

# Count number of data records.

for line in source:

if line.startswith('#-'): # Number of records

trash, expected = line.split()

expected = int(expected)

if line.startswith('#'): # Skip comments.

pass

elif line.startswith('D'): # Skip title line.

pass

else:

number += 1

source.close()

print "Expected number of data records:", expected

print "Actual number of data records:", number

if (expected == None):

print "Pass"

elif (expected == number):

print "Pass"

else:

print "Fail"

source.close()

## Lists (if time)

Python has very good support for list processing.

**Run at the command-line**

python

machines = ['HAL', 'Colossus', 'Maria', 'Robocop']

print len(machines)

print len([])

print machines[1]

Many languages like C, Java, Python index from 0.

Legacy from 70s - easier for the computer. Makes no sense nowadays!

print machines[0]

print machines[len(machines)]

print machines[len(machines) - 1]

print machines[-1]

print machines[-2]

Lists are mutable.

machines[1] = 'Tron'

print machines

machines[4] = 'Robbie'

Lists are heterogeneous.

mixed = ['stuff', 1, 2.5, True]

Iteration

for machine in machines:

print machine

machine is assigned to values not indices.

del machines[2]

print machines

del is a function.

machines.remove('Tron')

print machines

remove is a method on machines.

Another method.

machines.append('ED209')

print machines

machines.append('ED209')

machines.append('ED209')

print machines.count('ED209')

print machines

print machines.index('Robocop')

machines.insert(1, 'Terminator')

print machines

machines.sort()

print machines

sort does not return a value. The sort is done in-place.

Similarly.

machines.reverse()

print machines

'HAL' in machines

'Hector' in machines

We can create lists of data.

range(5)

range(2, 5)

range(1, 10, 3)