Python scripting

CIS 2235 Linux System Administration

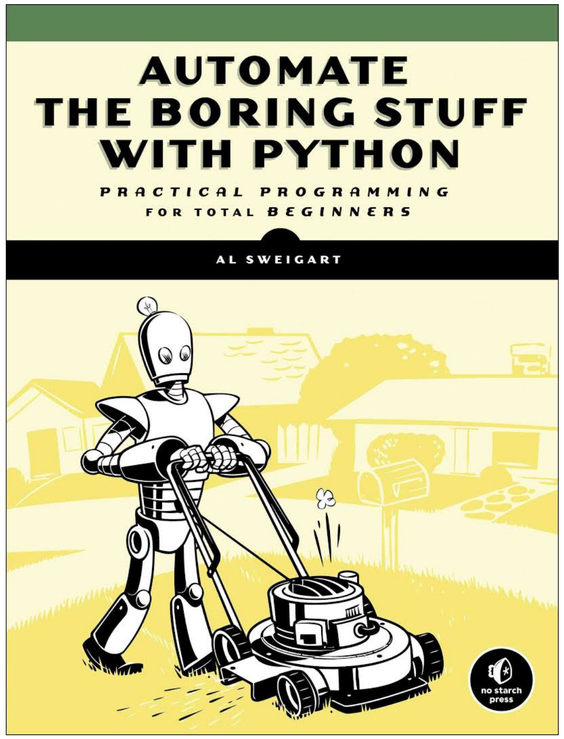
What Are Python Scripts?

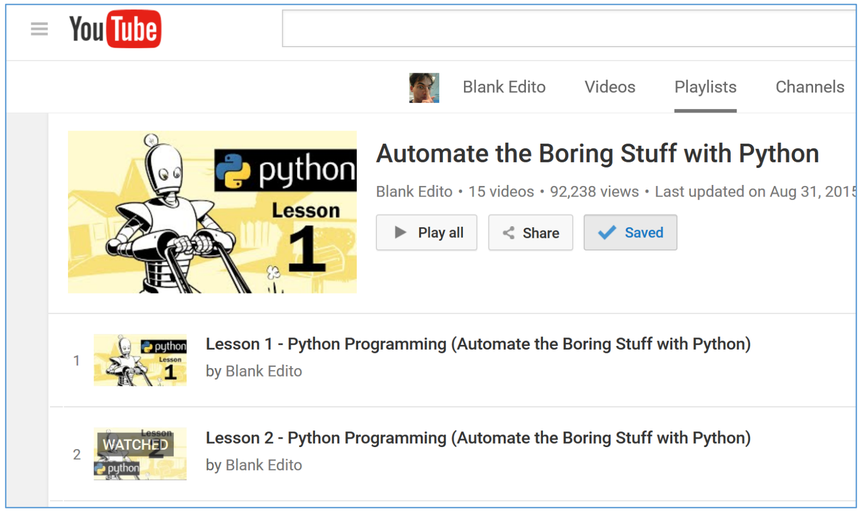
Scripting language: not compiled
Primarily used to automate daily, repeated tasks
Usually simple, but can get quite complicated
Perl and Python more powerful than bash
Perl is the ‘original’ and older popular scripting language
Python is the ‘new’
Google highly uses and supports it

Help sources

Good resource:
Released under Creative Commons License and available for free online:
Automate The Boring Stuff
He also has a video series.
The first 15 of 50 videos are on YouTube:
https://goo.gl/NRnn4s

- More general: Python tutorial





Environments

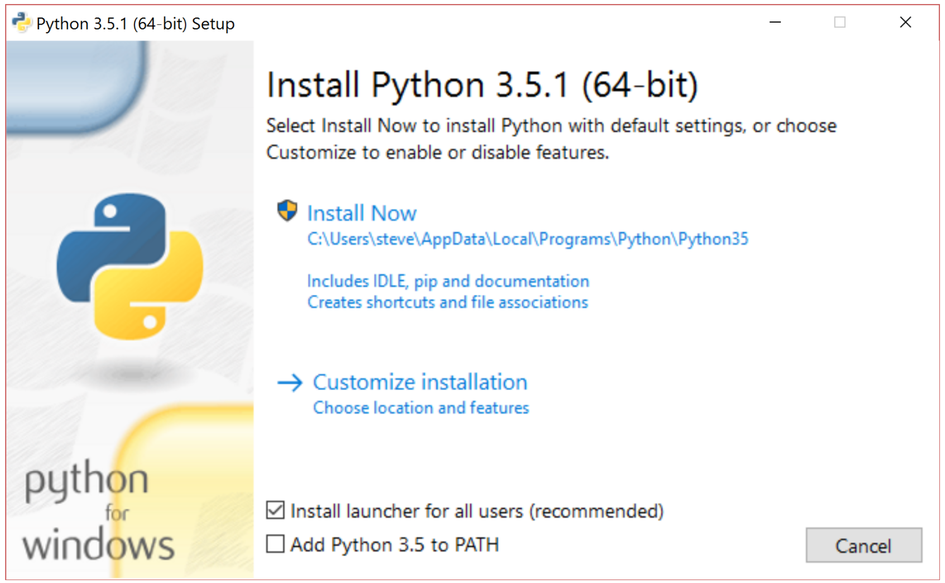
Linux

Python comes with Linux

$ python3

Windows/Mac OS X:

www.python.org



Be sure to run Python 3, not Python 2!

Python default IDE called idle

Idle

Interactive window

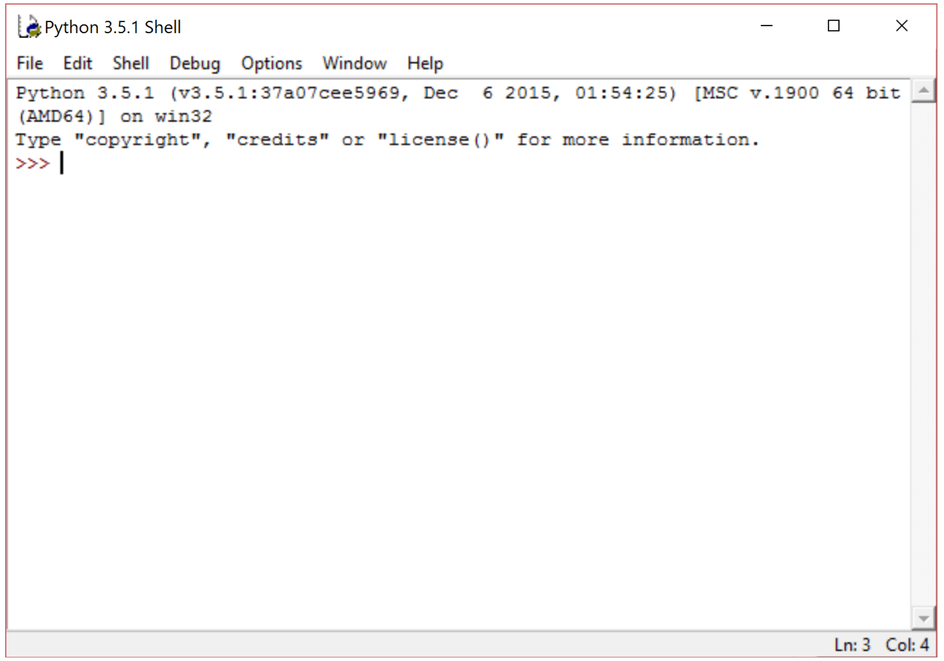
>>>

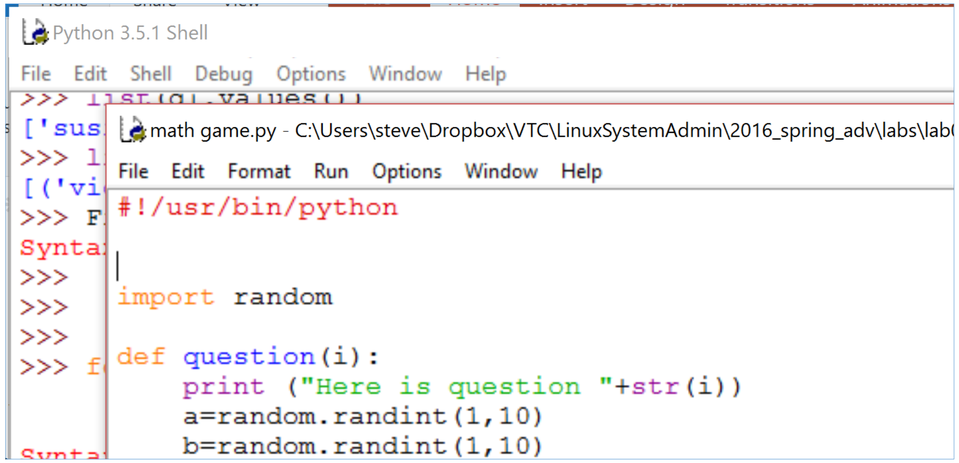
Great for playing with language and learning

We'll start here but quickly leave for a script

Within Idle, File → New to open a new editor.

If you want a fancier Python IDE, JetBrains Pycharm or Sublime. You can always use vi…





Key things to be able to do in Python

1. Variables: types, assigning, using
2. Start and print messages (“hello world”)
3. If-then-else (flow control & blocks of code)
4. Loops
5. Importing modules
6. Functions & scope
7. Lists & methods & tuples
8. File IO
9. Dictionaries

Initial stuff

Everything evaluates to a value

```
>>> 2+2
>>> (5 + 2) * ((3+2)/(4-2))
```

Everything has a data type

- Strings are quoted
- Ints and floats are different

```
>>> 2 + 'a' – not same type
>>> hello – unknown name hello
>>> 'hello' * 2 + 'world' works: 'hellohelloworld'
```

Variables (no special notation)

```
>>> spam = 42
```

Your first python script

shebang
Use comments (#)
Use the print command for stdout
Use '+' to concatenate strings together
Use input() to read in vars from user

```
#!/usr/bin/env python3
# my first python

print('hello world')
print('input a string: ')
mystr=input()
print('your string is ' + mystr )
```

Formatted printing

You cannot add numeric vars to a string.
Use str() to add numbers to a string

```
>>> x = 5
>>> print ("x = " + x)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: must be str, not int
>>> print ("x = " + str(x))
x = 5
>>>
```

- Another way with formatting:

```
>>> print ("x=%03d" % x)
x=005
>>>
```

Formatting in Python is done via the string formatting (`%`) operator:

```
"%02d:%02d:%02d" % (hours, minutes, seconds)
```

control flow

Comparison operators: ==, !=, <, <=, >, >=
True & False. >>> ‘a’ == ‘b’ or >>> 3==4

if syntax

```
if <condition>:
    <statements>
else:
    <statements>
<next statement>
```

Blocks of code

Blocks are defined by *indentation*

Python-aware editors are helpful here

Blocks of Code

Lines of Python code can be grouped together in *blocks*. You can tell when a block begins and ends from the indentation of the lines of code. There are three rules for blocks.

1. Blocks begin when the indentation increases.
2. Blocks can contain other blocks.
3. Blocks end when the indentation decreases to zero or to a containing block's indentation.

Blocks are easier to understand by looking at some indented code, so let's find the blocks in part of a small game program, shown here:

```
    if name == 'Mary':
❶       print('Hello Mary')
        if password == 'swordfish':
❷           print('Access granted.')
        else:
❸           print('Wrong password.')
```

While loops

while <condition> :
Indent next commands in loop

```
>>> while spam < 5:
        print ('spam=' + str(spam))
        spam = spam + 1

spam=0
spam=1
spam=2
spam=3
spam=4
>>>
```

For loops

```
for <var> in range() :
```

Indented statements in block

range(start, end, inc)

```
>>> for i in range(5):
        print ("var i is " + str(i))

var i is 0
var i is 1
var i is 2
var i is 3
var i is 4
>>>
```

Loop flow control

If a `break` command is encountered, the loop stops there and goes to the next block

If a `continue` command is encountered, the loop's flow immediately goes back to the top and starts the next iteration

```
>>> while True:
        print ("what is the secret password:")
        pw=input()
        if pw == 'please':
                break

what is the secret password:
slkdfj
what is the secret password:
444
what is the secret password:
please
>>> |
```

Importing modules

The commands so far have all been built-in from the "system" module.

To use other commands, "import" the module

```
import random
for i in range(5):
    print(random.randint(1, 10))
```

- Use the "from" version to not have to include module name

```
>>> from random import *
>>> for i in range(5):
        print(randint(10,20))

12
11
11
```

functions

Use "def" to define a new function (notice colon)
Pass local variables in list
Last value is returned, or use `return` command

```
import random
def getAnswer(answerNumber):
    if answerNumber == 1:
        return 'It is certain'
    elif answerNumber == 2:
        return 'It is decidedly so'
    elif answerNumber == 3:
        return 'Yes'
    else:
        return 'Reply hazy try again'

r = random.randint(1, 10)
fortune = getAnswer(r)
print(fortune)
```

Local scope

Variables in a function are only available in that function

```
def square(n=0):
    s = n*n

square(4)
```

```
>>> print (square(4))
16
>>> print ("s="+str(s))
Traceback (most recent call last):
  File "<pyshell#116>", line 1, in <module>
    print ("s="+str(s))
NameError: name 's' is not defined
>>> |
```

Lists

Perl calls these “arrays”, python calls them “lists”

Lists are defined with square brackets and comma separated elements

Each element can be referenced with square brackets → zero-based counting

Slices use colons: spam[0:2] are the first 3 elements

```
spam = ["cat", "bat", "rat", "elephant"]
```

spam[0] spam[1] spam[2] spam[3]

Working with lists

Append 2 lists with +

```
>>> ['a', 'b', 'c'] + [4, 5, 6]
['a', 'b', 'c', 4, 5, 6]
>>>
```

Delete with `del` command

```
>>> l1 = ['a', 'b', 'c'] + [4, 5, 6]
>>> l1
['a', 'b', 'c', 4, 5, 6]
>>> del l1[3]
>>> l1
['a', 'b', 'c', 5, 6]
>>>
```

Reassign:

```
>>> l1
['a', 'b', 'c', 5, 6]
>>> l1[0] = 'foobar'
>>> l1
['foobar', 'b', 'c', 5, 6]
>>>
```

Working with lists

Check if a var is an element in a list:

```
>>> l1
['foobar', 'b', 'c', 5, 6]
>>> if 'c' in l1:
        print ("got it")

got it
>>>
```

Get length of list: `len(<list>)`

Looping over list → `for <var> in <list>:`

```
>>> l1=['a','cat','dog',43,44]
>>> for i in l1:
        print ("the next value is "+str(i))

the next value is a
the next value is cat
the next value is dog
the next value is 43
the next value is 44
>>>
```

List methods

Same thing as a function, but it is "called on" a particular value.
The `index()` method returns the element position within a list.

- It is "called on" an array by "attaching" the method after the list variable.
- Other *list* methods:
  - append(), remove(), sort(), insert()
- Note: the method actually *changes* that list; it does <u>not</u> copy and return an edited version.

```
>>> l1.index("b")
1
>>> l1.index(5)
3
>>> l1
['foobar', 'b', 'c', 5, 6]
>>> |
```

```
>>> l1
['a', 'b', 'c']
>>> l1.append('foobar')
>>> l1
['a', 'b', 'c', 'foobar']
>>> |
```

String methods

These do not change the string, but return the edited string. So, you need to reassign.

upper(), lower()

isupper() islower() → (True/False)

join(), split()

strip(), lstrip(), rstrip()

```
>>> foobar='HELLO'
>>> foobar.lower()
'hello'
>>> foobar
'HELLO'
>>> foobar=foobar.lower()
>>> foobar
'hello'
>>>
```

Tuples

Look like lists (arrays) but have 1 major difference:

- Lists are mutable – can be changed
- Tuples are immutable – cannot be changed

Tuples are defined by parentheses

```
>>> foobar=('eggs','bacon',42,0.5)
>>> foobar
('eggs', 'bacon', 42, 0.5)
>>> foobar[1]
'bacon'
>>> foobar[1]='cheese'
Traceback (most recent call last):
  File "<pyshell#169>", line 1, in <module>
    foobar[1]='cheese'
TypeError: 'tuple' object does not support item assignment
>>>
```

Lists and tuples are ‘references’

When you set a new variable to a list or tuple, it is a reference, and both vars are *pointing to* the same memory location.
Same for functions
To avoid that, use copy() method.

```
>>> l1
['foobar', 'b', 'c', 5, 6]
>>> l2=l1
>>> l2[1] = 'cheese'
>>> l1
['foobar', 'cheese', 'c', 5, 6]
>>>
```

```
>>> def change2(array):
        array[2]='spam'

>>> change2(l1)
>>> l2
['foobar', 'cheese', 'spam', 5, 6]
>>>
```

File IO

`import os` → very helpful for getting paths and traversing dirs in the current OS

pwd: `os.getcwd()`

cd: `os.chdir()`

Splitting:

`os.path.dirname(), os.path.basename()`

ls: `os.listdir()`

Checks:

`os.path.exists(), os.path.isdir(), os.path.isfile()`

File IO

As normal: open, read, close file.
However, below is the "pythonic" method

```
with open(...) as f:
    for line in f:
        <do something with line>
```

The `with` statement handles opening and closing the file, including if an exception is raised in the inner block. The `for line in f` treats the file object `f` as an iterable, which automatically uses buffered IO and memory management so you don't have to worry about large files.

```
>>> with open("foobar") as f:
        for line in f:
                line=line.rstrip()
                print ("line: "+line)

line: foobar
line: foo
line: bar
line: eggs
```

File IO : write to a file

Open in ‘w’ mode
Use write() method on the file handle
Be warned, ‘w’ will truncate an existing file!
Close

```
>>> txt=open('foobar2','w')
>>> txt.write('this is a test\n')
15
>>> txt.write('foobar line\n')
12
>>> txt.close()
>>>
```

foobar2 - Notepad

File Edit Format View Help

```
this is a test
foobar line
```

Regex in python

1. `import re`
2. Compile the regex expression
3. Then use the search() method. The search() returns an object
4. Use the group() method to return the matched string

```
>>> import re
>>> pn=re.compile(r'\d{3}[ -]\d{3}[- ]\d{4}')
>>> pn.search("my phone is 802-555-1212.")
<_sre.SRE_Match object; span=(12, 24), match='802-555-1212'>
>>> phonenum1 = pn.search("joe blow 555-444-2211")
>>> print ("num: "+ phonenum1.group())
num: 555-444-2211
>>> |
```

Dictionaries

Key-value pairs!
Coded with curley-brackets {}
Used to 'lookup' a value from a key. (fast)
Keys must be unique (of course)
Hardcode: var={k1:v1, k2:v2, ... }
Use: var[key] returns the value

```
>>> d1 = { 'president':'bob','vice':'susie','treas':'sally','sec':'joe'}
>>> d1['sec']
'joe'
>>> |
```

Lists from dictionaries

List of keys is usually most important
We often loop over them.
Using the dict in a for loop, returns all the keys

```
>>> for k in d1:
        print ("k="+k, end='')
        print (";  v="+d1[k])


k=vice;  v=susie
k=treas;  v=sally
k=sec;  v=joe
k=president;  v=bob
>>> |
```

More lists from dictionaries

.keys(), .values(), .items()

These are not true lists. Use list() function.

```
>>> list(d1.keys())
['vice', 'treas', 'sec', 'president']
>>> list(d1.values())
['susie', 'sally', 'joe', 'bob']
>>> list(d1.items())
[('vice', 'susie'), ('treas', 'sally'), ('sec', 'joe'), ('president', 'bob')]
>>> |
```

- Loop shortcut:

```
>>> for k,v in d1.items():
        print ("k="+k+" -->  v="+v)

k=vice -->  v=susie
k=treas -->  v=sally
k=sec -->  v=joe
k=president -->  v=bob
>>> |
```