Hands-on Beginning Python

@__mharrison__
http://hairysun.com

About Me

- 12 years Python
- Worked in HA, Search, Open Source, BI and Storage
- Author of multiple Python Books

Get code

`beg_python.tar.gz`

- Thumbdrive has it

Unzip it somewhere (`tar -zxvf beg_python.tar.gz`)

Begin

Warning

- Starting from zero
- Hands on
  - (short) lecture
  - (short) code
  - repeat until time is gone

Why Python?

- Used (almost) everywhere
- Fun
- Concise

Python 2 or 3?

Most of this is agnostic. I'll note the differences, but use 2.x throughout

Hello World

hello world

```
print "hello world"
```

from interpreter

```
$ python
>>> print "hello world"
hello world
```

REPL

```
$ python
>>> 2 + 2   # read, eval
4           # print
>>>         # repeat (loop)
```

REPL (2)

Many developers keep a REPL handy during programming

From script

Make file `hello.py` with

```
print "hello world"
```

Run with:

```
python hello.py
```

(unix) script

Make file `hello` with

```
#!/usr/bin/env python
print "hello world"
```

Run with:

```
chmod +x hello
./hello
```

Python 3 hello world

`print` is no longer a statement, but a function

```
print("hello world")
```

Objects

Objects

Everything in *Python* is an object that has:

- an *identity* (`id`)
- a *type* (`type`). Determines what operations object can perform.
- a *value* (mutable or immutable)

id

```
>>> a = 4
>>> id(a)
6406896
```

type

```
>>> a = 4
>>> type(a)
<type 'int'>
```

Value

- **Mutable:** When you alter the item, the `id` is still the same. Dictionary, List
- **Immutable:** String, Integer, Tuple

Mutable

```
>>> b = []
>>> id(b)
140675605442000
>>> b.append(3)
>>> b
[3]
>>> id(b)
140675605442000    # SAME!
```

Immutable

```
>>> a = 4
>>> id(a)
6406896
>>> a = a + 1
>>> id(a)
6406872   # DIFFERENT!
```

Variables

```
a = 4              # Integer
b = 5.6            # Float
c = "hello"        # String
a = "4"      # rebound to String
```

Naming

- lowercase
- underscore_between_words
- don't start with numbers

See PEP 8

Example Assignment

`sample.py`

Assignment

`variables.py`

Math

`+`, `-`, `*`, `/`, `**` (power), `%` (modulo)

Careful with integer division

```
>>> 3/4
0
>>> 3/4.
0.75
```

(In Python 2, in Python 3 `//` is integer division operator)

What happens when you raise 10 to the 100th?

Long

```
>>> 10**100
100000000000000000000000000000000
000000000000000000000000000000000
000000000000000000000000000000000
00L
```

Strings

```
name = 'matt'
with_quote = "I ain't gonna"
longer = """This string has
multiple lines
in it"""
```

String formatting

c-like

```
>>> "%s %s" %('hello', 'world')
'hello world'
```

PEP 3101 style

```
>>> "{0} {1}".format('hello',
'world')
'hello world'
```

dir

```
>>> dir("a string")
['__add__', '__class__', ...
'startswith', 'strip',
'swapcase', 'title', 'translate',
'upper', 'zfill']
```

Whats with all the `'__blah__'`?

dunder methods

dunder (double under) or "special/magic" methods determine what will happen when `+` (`__add__`) or `/` (`__div__`) is called.

help

```
>>> help("a string".startswith)
```

```
Help on built-in function startswith:

startswith(...)
  S.startswith(prefix[, start[, end]]) -> bool

  Return True if S starts with the specified prefix, False
otherwise.
  With optional start, test S beginning at that position.
  With optional end, stop comparing S at that position.
  prefix can also be a tuple of strings to try.
```

Assignment

`strings.py`

Comments

comments

Comments follow a #

More Types

None

Pythonic way of saying NULL. Evaluates to False.

```
c = None
```

booleans

```
a = True
b = False
```

sequences

- lists
- tuples
- sets

lists

```
>>> a = []
>>> a.append(4)
>>> a.append('hello')
>>> a.append(1)
>>> a.sort() # in place
>>> print a
[1, 4, 'hello']
```

tuples

Immutable

```
>>> b = (2,3)
>>> b.append(5)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'tuple' object has no attribute
'append'
```

tuple vs *list*

- Tuple
  - Hetergenous state (name, age, address)
- List
  - Homogenous, mutable (list of names)

Assignment

`lists.py`

Dictionaries

dictionaries

Also called *hashmap* or *associative array* elsewhere

```
>>> age = {}
>>> age['george'] = 10
>>> age['fred'] = 12
>>> age['henry'] = 10
>>> print age['george']
10
```

dictionaries (2)

Find out if `'matt'` in `age`

```
>>> 'matt' in age
False
```

.get

```
>>> print age['charles']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'charles'
>>> print age.get('charles', 'Not found')
Not found
```

.setdefault

```
>>> if 'charles' not in age:
...     age['charles'] = 10
```

shortcut

```
>>> age.setdefault('charles', 10)
10
```

`.setdefault` (2)

- set the value for key if missing
- also returns the value

.setdefault(3)

Even more useful if we map to a list of items

```
>>> room2members = {}
>>> member = 'frank'
>>> room = 'room5'
>>> room2members.setdefault(room,
[]).append(member)
```

.setdefault (4)

Even more useful if we map to a list of items

```
>>> room2members = {}
>>> member = 'frank'
>>> room = 'room5'
>>> if room in room2members:
...     members = room2members[room]
...     members.append(member)
... else:
...     members = [member]
...     room2members[room] = members
```

.setdefault(5)

collections.defaultdict is probably a better choice

```
>>> from collections import defaultdict
>>> room2members = defaultdict(list)
>>> member = 'frank'
>>> room = 'room5'
>>> room2members[room].append(member)
```

deleting keys

Removing 'charles' from age

```
>>> del age['charles']
```

Assignment

`dictionaries.py`

Functions

functions

```
def add_2(num):
    """ return 2
    more than num
    """
    return num + 2

five = add_2(3)
```

whitespace

Instead of { use a : and indent consistently (4 spaces)

whitespace (2)

invoke `python -tt` to error out during inconsistent tab/space usage in a file

default (named) parameters

```
def add_n(num, n=3):
    """default to
    adding 3"""
    return num + n

five = add_n(2)
ten = add_n(15, -5)
```

`__doc__`

Functions have *docstrings*. Accessible via `.__doc__` or `help`

__doc__

```
>>> def echo(txt):
...     "echo back txt"
...     return txt
>>> help(echo)
Help on function echo in module __main__:
<BLANKLINE>
echo(txt)
    echo back txt
<BLANKLINE>
```

naming

- lowercase
- underscore_between_words
- don't start with numbers
- verb

See PEP 8

Assignment

`functions.py`

Conditionals

conditionals

```
if grade > 90:
    print "A"
elif grade > 80:
    print "B"
elif grade > 70:
    print "C"
else:
    print "D"
```

Remember the colon/whitespace!

Booleans

```
a = True
b = False
```

Boolean tests

Supports (>, >=, <, <=, ==, !=)

```
>>> 5 > 9
False
>>> 'matt' != 'fred'
True
>>> isinstance('matt',
basestring)
True
```

Iteration

iteration

```
for number in [1,2,3,4,5,6]:
    print number

for number in range(1, 7):
    print number
```

iteration (2)

Java/C-esque style of object in array access (BAD):

```
animals = ["cat", "dog", "bird"]
for index in range(len(animals)):
    print index, animals[index]
```

iteration (3)

If you need indices, use `enumerate`

```
animals = ["cat", "dog", "bird"]
for index, value in enumerate(animals):
    print index, value
```

iteration (4)

Can `break` out of nearest loop

```
for item in sequence:
    # process until first negative
    if item < 0:
        break
    # process item
```

iteration (5)

Can `continue` to skip over items

```
for item in sequence:
    if item < 0:
        continue
    # process all positive items
```

iteration (6)

Can loop over lists, strings, iterators, dictionaries... sequence like things:

```
my_dict = { "name": "matt", "cash": 5.45}
for key in my_dict.keys():
   # process key

for value in my_dict.values():
   # process value

for key, value in my_dict.items():
   # process items
```

pass

pass is a null operation

```
for i in range(10):
    # do nothing 10 times
    pass
```

Hint

Don't modify *list* or *dictionary* contents while looping over them

Assignment

`loops.py`

Slicing

Slicing

Sequences (lists, tuples, strings, etc) can be *sliced* to pull out a single item

```
my_pets = ["dog", "cat", "bird"]
favorite = my_pets[0]
bird = my_pets[-1]
```

Slicing (2)

Slices can take an end index, to pull out a list of items

```
my_pets = ["dog", "cat", "bird"]
# a list
cat_and_dog = my_pets[0:2]
cat_and_dog2 = my_pets[:2]
cat_and_bird = my_pets[1:3]
cat_and_bird2 = my_pets[1:]
```

Slicing (3)

Slices can take a stride

```
my_pets = ["dog", "cat", "bird"]
# a list
dog_and_bird = [0:3:2]
zero_three_etc = range(0,10)[::3]
```

Slicing (4)

Just to beat it in

```
veg = "tomatoe"
correct = veg[:-1]
tmte = veg[::2]
oetamot = veg[::-1]
```

File IO

File Input

Open a file to read from it (old style):

```
fin = open("foo.txt")
for line in fin:
    # manipulate line

fin.close()
```

File Output

Open a file using `'w'` to write to a file:

```
fout = open("bar.txt", "w")
fout.write("hello world")
fout.close()
```

Always remember to close your files!

closing with `with`

implicit `close` (new 2.5+ style)

```
with open('bar.txt') as fin:
    for line in fin:
        # process line
```

Hint

Much code implements "file-like" behavior (`read`, `write`). Try to use interfaces that take files instead of filenames where possible.

Hint (2)

```
def process_fname(filename):
    with open(filename) as fin:
        process_file(fin)


def process_file(fin):
    # go crazy
```

Classes

Classes

```
class Animal(object):
    def __init__(self, name):
        self.name = name

    def talk(self):
        print "Generic Animal Sound"

animal = Animal("thing")
animal.talk()
```

Classes (2)

notes:

- `object` (base class) (fixed in 3.X)
- *dunder* init (constructor)
- all methods take `self` as first parameter

Classes(2)

Subclassing

```
class Cat(Animal):
    def talk(self):
        print '%s says, "Meow!"' % (self.name)

cat = Cat("Groucho")
cat.talk() # invoke method
```

Classes(3)

```
class Cheetah(Cat):
    """classes can have
    docstrings"""

    def talk(self):
        print "Growl"
```

naming

- CamelCase
- don't start with numbers
- Nouns

Assignment

`classes.py`

Packages, Modules and Importing

PYTHON_PATH

`PYTHON_PATH` (env variable) and `sys.path` determine where Python looks for packages and modules.

importing

Python can `import` *packages* and *modules* via:

```
import package
import module
from math import sin
import longname as ln
```

Packages

Any directory that has a `__init__.py` file in it and is in `PYTHONPATH` is importable.

Packages (2)

File layout (excluding README, etc). Note the `__init__.py` file in package directories:

```
packagename/
   __init__.py
   code1.py
   code2.py
   subpackage/
      __init__.py
```

Packages (3)

```
import packagename.code2
from packagename import code1

packagename.code2.bar()
code1.baz()
```

Modules

Just a `.py` file in a directory in PYTHONPATH

Modules (2)

```
import modulename

modulename.foo()
```

naming

- lowercase
- nounderscorebetweenwords
- don't start with numbers

to install code

- via operating system
- `easy_install`
- updating `PYTHONPATH` (env variable)
- changing `sys.path` in client code

Exceptions

Exceptions

Can catch exceptions

```
try:
      f = open("file.txt")
except IOError, e:
      # handle e
```

Exceptions (2)

2.6+/3 version (as)

```
try:
    f = open("file.txt")
except IOError as e:
    # handle e
```

Exceptions (3)

Can raise exceptions

```
raise RuntimeError("Program failed")
```

Chaining Exceptions (3)

```
try:
    some_function()
except ZeroDivisionError, e:
    # handle specific
except Exception, e:
    # handle others
```

finally

`finally` always executes

```
try:
    some_function()
except Exception, e:
    # handle others
finally:
    # cleanup
```

else

runs if no exceptions

```
try:
    print "hi"
except Exception, e:
    # won't happen
else:
    # first here
finally:
    # finally here
```

re-raise

Usually a good idea to re-raise if you don't handle it. (just `raise`)

```
try:
    # errory code
except Exception, e:
    # handle higher up
    raise
```

some hints

- try to limit size of contents of `try` block.
- catch specific Exceptions rather than just `Exception`

File Organization

program layout

```
PackageName/
  README
  setup.py
  bin/
    script
  docs/
  test/ # some include in package_name
  packagename/
    __init__.py
    code1.py
    subpackage/
      __init__.py
```

scripts vs libraries

- Executable?
- Importable?

script layout

- #!/usr/bin/env python
- docstrings
- importing
- meta data
- logging
- implementation
- testing?
- `if __name__ == '__main__':`
- optparse (now argparse)

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

docstrings

Module level docstrings at top of file are visible at `module.__doc__` or `help(module)`

importing

Group by:

- stdlib packages
- local packages
- 3rd party packages

importing (2)

```
import deeply.nested.package
...
bar = deeply.nested.package.Bar()
```

or

```
from deeply.nested.package import Bar
...
bar = Bar()
```

metadata

```
VERSION = (0, 2, 1)
```

or

```
__version__ = '0.2.1'
```

ifmain

```
if __name__ == '__main__':
    sys.exit(main(sys.argv))
```

main

```
def main(arguments):
    # process args
    # run
    # return exit code
```

What if I want to reuse logic from my script?

Script wrapper

Put logic in package or module and have script be a simple wrapper. Script would look like this:

```
#!/usr/bin/env python
import sys
import scriptlib
sys.exit(scriptlib.main(sys.argv)
)
```

Debugging

Poor mans

`print` works a lot of the time

Remember

Clean up `print` statements. If you really need them, use `logging` or write to `sys.stdout`

pdb

```
import pdb; pdb.set_trace()
```

pdb commands

- `h` - help
- `s` - step into
- `n` - next
- `c` - continue
- `w` - where am I (in stack)?
- `l` - list code around me

Testing

testing

see `unittest` and `doctest`

nose

nose is useful to run tests with coverage, profiling, break on error, etc

3rd party

Packaging

packaging

Somewhat of a mess and in flux. Find something else that does what you want and steal.... er ....copy it.

packaging (2)

- **`virtualenv`**

  manage different python environments

- **`distribute`**

  install code

- **`pip`**

  manage code in environment

3rd party

packaging (3)

pypi hosts packages

Other Tools

Editors

Most editors have some notion of Python support

Linting

- `pyflakes`- least verbose (dead/redundant code)
- `pychecker` - more verbose, imports code, slower
- `pylint` - most verbose, configurable, "rates" code

3rd party

Refactoring

- `rope` - not perfect, somewhat slow

3rd party

That's all

Questions? Tweet or email me

matthewharrison@gmail.com
@__mharrison__
http://hairysun.com