Hands-on Beginning Python

@__mharrison__
http://panela.blog-city.com/

Purpose

Cram 11 years of experience into 3 hours. You should walk out of here being able to write and read (most) Python.

Warning

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

Get code

`beg_python.tar.gz` or `beg_python.zip` if you prefer

- Thumbdrive has it
- Also at http://panela.blog-city.com/

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

About Me

- 11 years Python
- Worked at various startups in high availability, search, open source, business intelligence and storage

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

Python installation

- Already installed on most unices
- `python-2.7.2.msi` on drive

Python editors

There are many:

- emacs, vim
- eclipse (py-dev), jetbrains (PyCharm)
- Wing, Komodo
- ...
- IDLE (bundled with Python)

Interactive Interpreter

Interpreter

Contains REPL

- Read
- Evalute
- Print
- Loop

Interpreter (2)

Where you see `>>>` type in what follows
(`...` is indented continuation)

Interpreter (3)

```
$ python
Python 2.6.6 (r266:84292, Feb
26 2011, 23:10:42)
[GCC 4.3.4] on linux2
Type "help", "copyright",
"credits" or "license" for
more information.
>>>
```

Interpreter (4)

Hello World from the interpreter

```
>>> print "hello world"
hello world
```

Interpreter (6)

Python 3 version

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

Interpreter (7)

Many developers keep open at all times

- introspection
- testing (`doctest`)
- experimenting

Running Python Programs

Program

Create file `hello.py`:

```
print "hello world"
```

Program (2)

Run it:

```
$ python hello.py
```

Program (3)

Unix tweaks, edit file `hello.py`:

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

Flip on executable bit:

```
$ chmod +x hello.py
```

Program (4)

Run it:

```
$ ./hello.py
```

Assignment

Run `hello.py` on your laptop

Variables

Variables

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

naming

- lowercase
- underscore_between_words
- do not start with numbers
- do not use built-ins (`dir`, `file`, etc)
- do not use keywords (`assert`, `class`, `def`, etc)

See PEP8

Example Assignment

`sample.py`

Assignment

`variables.py`

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 = 5
>>> id(a)
6406872 # DIFFERENT!
```

Assignment

`objects.py`

Numbers

Integers and Floats

Python has native support

```
>>> num = 2          # integer
>>> other = 3.4      # float
```

Casting

```
>>> num = '2.3'
>>> float(num)
2.3
>>> int('2')
2
>>> long('2')
2L
```

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
10000000000000000000000000000
00000000000000000000000000000
00000000000000000000000000000
00000000000000L
```

Long (2)

```
>>> long1 = 1L
>>> num = 3
>>> long3 = long(3)
```

Assignment

`numbers.py`

Strings

Strings

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

Escape sequence

Use \ (backslash) to escape

```
>>> aint = 'ain\'t'
>>> bslash = '\\'
>>> newline = '\n'
```

String formatting

c-like

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

PEP 3101 style

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

String methods

Strings are objects and object have *methods* you can call on them

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.
```

Common string methods

endswith

```
>>> xl = 'Oct2000.xls'
>>> xl.endswith('xls')
True
```

(also `startswith`)

find

```
>>> word = 'grateful'
>>> word.find('ate')  # 0 is g, 1 is r, 2 is a
2
>>> word.find('great')
-1
```

format

```
>>> print 'name: {0}, age: {1}'.format('Matt', 10)
name: Matt, age: 10
```

join

```
>>> ', '.join(['1','2','3'])
'1, 2, 3'
```

strip

```
>>> '   hello  there
'.strip()
'hello  there'
```

(also `lstrip`, `rstrip`)

Assignment

`strings.py`

Comments, Booleans and None

Comments

Comments follow a #

```
>>> # This line is ingored
>>> num = 3.14  # almost Pi
```

Comments (2)

Do not use triple quoted strings as a substitue for lack of multi-line comments.

Booleans

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

None

Pythonic way of saying NULL. Evaluates to False.

```
c = None
```

Truthy/Falsey

```
>>> bool(0)
False
>>> bool(4)
True
>>> bool('')
False
>>> bool('0')  # The string containing 0,
not the number zero
True
```

Truthy/Falsey (2)

True:

- `True`
- Non-zero numbers
- Non-empty strings
- Non-empty collections
- Most objects

Truthy/Falsey (3)

False:

- `False`
- `None`
- `0`
- `[]`, `{}`

Conditionals

conditionals

Python supports checking for equality, greater than, less than, etc.

(`==`, `!=`, `>`, `>=`, `<`, `<=`, `is`, `is not`)

`==` vs `is`

`is` checks identity (same objects), `==` checks for equality (perhaps different, but with same value)

is

Usually used for checking against `None`

```
>>> if value is not None:
...     # initialize value
```

Boolean operators

Used to combine conditional logic

- `and`
- `or`
- `not`

`if` statements

```
if grade > 90 and grade <= 100:
    print "A"
elif grade > 80:
    print "B"
elif grade > 70:
    print "C"
else:
    print "D"
```

`if` statements (2)

Can optionally have multiple `elif` (else if) statements. One optional `else` statement.

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

Sequences

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 hints

Gotchas

```
>>> normal = (2,3,4)
>>> single = (3,)   # note comma
>>> three = (3)
>>> three           # an integer!
3
>>> empty = ()      # no comma
```

tuple vs *list*

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

range

Useful built-in function

```
>>> range(5)  # numbers from zero to (not including) 5
[0, 1, 2, 3, 4]
```

Assignment

`lists.py`

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`

Dictionaries

dictionaries

Also called *hashmap* or *associative array*

```
>>> 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
```

removal

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

Assignment

`functions.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]
```

Assignment

`slicing.py`

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
# fin closed here
```

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`

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`, see `help` for `exceptions` module

Creating Exceptions

Subclass `Exception` (or appropriate subclass):

```
class MyException(Exception):
    """ badness occurred """
```

Assignment

`error.py`

Using other code

PYTHONPATH

`PYTHONPATH` (env variable) and `sys.path` (Python variable) 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
```

Star imports

Do not do this!

```
from math import *
```

Nested libraries

```
>>> import xml.etree.ElementTree
>>> elem =
xml.etree.ElementTree.XML('<xml><foo/></xml>')
>>> from xml.etree.ElementTree import XML
>>> elem2 = XML('<xml><foo/></xml>')
```

Import organization

Group by:

- Standard library imports
- 3rd party imports
- Local package imports

See PEP8

Import example

```
#!/usr/bin/env python
""" This module does foo...  """
import json          # standard libs
import sys

import psycopg2      # 3rd party lib

import foolib        # local library
```

Finding Libraries

- "Batteries included"
- PyPi (Package Index)

Assignment

`importing.py`

Creating libraries (Modules and Packages)

Modules

Just a Python (`modulename.py`) file in a directory in `PYTHONPATH`

Modules (2)

```
import modulename

modulename.foo()
```

Modules (3)

Modules can have docstrings. Must be first line of file (or following shebang).

naming

- lowercase
- nounderscorebetweenwords
- don't start with numbers

See PEP 8

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()
```

naming

- lowercase
- short
- no underscores
- don't start with numbers

See PEP 8

Assignment

`libraries.py`

`pycat` an example

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 must be top of file (below shebang). They are visible at `module.__doc__` or `help(module)`.

docstrings (2)

```
r"""A simple implementation the the unix ``cat`` command. It only
implements the ``--number`` option. It is useful for illustrating
file layout and best practices in Python.

This is a triple quoted docstring for the whole module (this
file). If you import this module somewhere else and run
``help(cat)``, you will see this.

This docstring also contains a ``doctest`` which serves as an
example of programmatically using the code. It also serves as
tests, since the ``doctest`` module can execute this test and
validate it by checking any output.
```

docstrings (3)

```
>>> import StringIO
>>> fin = StringIO.StringIO('hello\nworld\n')
>>> fout = StringIO.StringIO()
>>> cat = Catter([fin], show_numbers=True)
>>> cat.run(fout)
>>> print fout.getvalue()
     0  hello
     1  world
<BLANKLINE>
"""
```

importing

Group by:

- stdlib packages
- local packages
- 3rd party packages

importing (2)

```
import argparse
import logging
import sys
```

metadata

```
__version__ = '0.2.1'
```

See PEP 386 and 396

logging

```
logging.basicConfig(level=log
ging.DEBUG)
```

Logic

`Catter` class

ifmain

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

main

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

program layout

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

Installation of Packages

Types of install

- Global install (root) - system packages
- Create virtual environment
- Mangle `PYTHONPATH`

packaging

- **`virtualenv`**

  manage different python environments

- **`distribute`**(**`easy_install`**)install

  code

- **`pip`**

  manage code in environment

3rd party

packaging (2)

`pypi` hosts packages

`virtualenv`

Let's you handle different version/dependencies.

`virtualenv` (2)

Installation via system or `easy_install virtualenv`

`virtualenv` (3)

Create an environment:

```
$ virtualenv myenv
```

Creates

```
path/to/myenv/lib/python2.x/site-packages
```

`virtualenv` (4)

Activate an environment:

```
$ source myenv/bin/activate
```

`$PATH` now has `path/to/myenv/bin` in it. `python` runs from that directory.

`virtualenv` (5)

`virtualenv` installs `pip` during creation of environment

`pip`

Handles installing/uninstalling packages

`pip` (2)

To install packages:

```
$ pip install SomePackage
```

`pip` (3)

Create requirements file (`req.txt`):

```
SomePackage
Django==1.3
xlwt>=0.6.8
```

Install them:

```
$ pip install -r req.txt
```

`pip` (4)

Get requirements from environment:

```
$ pip freeze > req.txt
```

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

Input

Input

Useful for learning

```
name = input('enter name:')
```

see also `getpass` module

One last thing

That's all

Questions? Tweet or email me

matthewharrison@gmail.com
@__mharrison__
http://panela.blog-city.com/