Exploring python with a background in Java

Andreas Grivas[1]

[1] Informatics and Telematics
Harokopeio University of Athens

April 25, 2015

Summary

Disclaimers

self.disclaimers

- No expertise in python is implied
- Approaches reflect personal taste and may or may not be **pythonic**
- Will cover features I consider useful

We are here ..

to

- Understand the **basics**
- See **practical** python examples
- Stress **philosophy**

and not to

- Become technical
- Compare python libraries and tools

Python info

```
# incompatibility example
print 'hello world'   # python 2
print('hello world') # python 3
```

- A laconic programming language that emphasizes readability
- Implementation started in December 1989 by Guido van Rossum
- Written in C
- Current stable versions 2.7.9 & 3.4.3
- Python3 is backwards incompatible with Python2
- **Open source - https://github.com/python-git/python**

Python is..

```
# hello python in line comments!

message = 'hello python'    # 1. I didn't say String message
print(message + 5)          # 2. Error!
print(message + str(5))     # 2. Ok.
print('num :' + str(5))     # 3. I could run this line of code alone
def my_favourite_number(): # 4. I can define a function not in a class
    return -32
class FavouriteNumber:      # 4. I could use a class
    num = -32
```

1. **Dynamically typed** - no variable types (think javascript)
2. **Strongly typed** - no guessing if not well defined (don't think javascript)
3. Feels **interpreted** - can run one line of code [1] (think octave)
4. Supports many **programming paradigms** (don't think Java)
   1. **Procedural** - your building blocks are functions (think C)
   2. **ObjectOriented** - your building blocks are classes (think Java)
   3. More..

[1] Depends on implementation of python (CPython, Jython, IronPython, Stackless

Philosophy

Python is a loose, programmer friendly language that is based on good intentions.
It supposes you will play nicely with other programmers and use your freedoms wisely.
It doesn't impose restrictions supposing that you are intelligent enough not to shoot your own foot just because you were given a gun.

Why do you say that?

- Use programming paradigm that matches the situation
- Don't have to capture exceptions
- Operator overloading
- No encapsulation

IPython

We will be exploring python with ipython, python3 and an editor if needed.

Useful links

- **Python Tutor - visualize code execution**
- **IPython notebooks - check these out**
- **IPython website**

Variables

definition examples

```
# Basic variables example
# with terrible choices for names.
# as we said before - no types

# basic types
b = True     # bool
i = 5        # int
f = 5.0      # float
s = 'hi'     # str

# built in data structures
# iterables
t = ()       # empty tuple
l = []       # empty list
d = {}       # empty dictionary
s = set()    # empty set
```

naming examples

```
# Naming examples
# More important to have good names
# because of no type

# variable Java style - CamelCase
randomInteger = 5

# variable C style
random_integer = 5

# classes start uppercase
class BeerGenerator:
    pass

# functions underscored with lowercase
def spawn_food(location):
    pass
```

Printing

3 ways to print

```
# I know 3 ways

name = 'Jeremiah'
short_pie = 3.1415

# C style formatted
print('hello %s, have some %.2f' % (name, short_pie))

# use format
print('hello {}, have some {}'.format(name, short_pie))

# Java style using + operator
print('hello ' + name + ', have some ' + str(short_pie))
```

Imports

using libraries

```
# use a python library
# below examples are equivalent

# include the random module in namespace
import random
rand = random.randrange(10)

# include random with alias
import random as rnd
rand = rnd.randrange(10)

# include target object from module
from random import randrange
rand = randrange(10)

# include target object only with alias
from random import randrange as rrange
rand = rrange(10)
```

Example

script

```
# 1st script
from random import randrange      # randrange returns a random number 0 - to target - 1
dice_roll = randrange(6) + 1      # get a dice roll - could have done it: randrange(1,7)
character_roll = randrange(26)    # 26 letters in the alphabet - pick a value out of 26
ascii_a_value = ord('a')          # get ASCII code of a
print(type(ascii_a_value))        # int
rand_char = chr(ascii_a_value + character_roll)  # pick a random char
print(type(rand_char))            # str - no char
message = rand_char * dice_roll   # what message is this? multiplication on string?
print(message)                    # print it!
```

If Else

what if..

```
# Decisions.. decisions..
# Goodbye curly braces {}!
# Indentation Warning - it matters!

using_tabs = True
if using_tabs:
    print('You are going to have a tough time -_-')
else:
    print('Whatever - it\'s not going to print')

# bonus - if you are just going to initialize something
programmer = 'Happy' if not using_tabs else 'Unhappy'
```

For - While

for.. starters

```
# for.. is boring like this - why later
# so.. here's a boring example

n = 10
for x in range(n):
   print(x)

print()            # leave a newline

x = n
# while
while(x>0):
    x -= 1         # sorry C guys - no x--
    if(x%2==0):
        continue   # don't print even numbers
    print(x)
```

Exceptions

try catch..

```
# Ah.. exceptions
# Note to java programmers!
# It's error not exception
# It's except and not catch
# It's raise not throw
# And don't overuse this feature

try:
    print('This gotcha will happen at least ' + 10 + ' more times')
except TypeError as e:
    print('I can\'t do that')
    print(e)
```

With

with .. as

```
# Safely exit construct - like a finally
# Even if an error occurs file will be closed
# Catching exception is meaningful if you can do something
# or you want to print a nicer message

filename = 'examples.txt'
with open(filename, 'r') as f:
    lines = f.readlines()
```

List

shopping..

```
# List is iterable - ordered
# List is mutable

shopping = list('spam', 'flour', 'spam', 'eggs')  # create a list

shopping = ['spam', 'flour', 'spam', 'eggs']      # equivalent

num_items = len(shopping)  # get size of list
print(shopping)            # print whole list
print(shopping[0])         # access first element and print it
shopping[0] = 'cookies'    # change first element
print(shopping[0])         # access changed element and print

# much better for example - no indexes
for item in shopping:
    print(item)

# if you really really need indexes
for index, item in enumerate(shopping):
    print('item %s in list: %s' % (index, item))

# list comprehension <3
no_spam = [item for item in shopping if item != 'spam']
```

Tuple

unchangeable list

```
# Tuple is iterable - ordered
# Tuple is immutable - can't change it after initialization

shopping = tuple('spam', 'flour', 'spam', 'eggs')  # create a tuple

shopping = ('spam', 'flour', 'spam', 'eggs')       # equivalent

num_items = len(shopping)  # get size of tuple
print(shopping)            # print whole tuple
print(shopping[0])         # access first element and print it
shopping[0] = 'cookies'    # oops.. no can do
print(shopping[0])         # well.. the same as before

# previous example also works - show reversed - these functions can be chained
for item in reversed(shopping):
    print(item)

# if you really really need indexes
for index, item in enumerate(shopping):
    print('item %s in list: %s' % (index, item))

# tuple comprehension <3
no_spam = [item for item in shopping if item != 'spam']
```

Set

no duplicates

```
# Set is iterable - unordered (can't access using [])
# Set is mutable

shopping = set('spam', 'flour', 'spam', 'eggs')  # create a set

shopping = {'spam', 'flour', 'spam', 'eggs'}     # equivalent

num_items = len(shopping)  # get size of set
print(shopping)            # print whole set
print(shopping[0])         # nope. set is unordered - no meaning in first element
print('spam' in shopping)  # with sets we usually check for membership
shopping[0] = 'cookies'    # oops.. no can do

# previous example also works - show reversed - these functions can be chained
for item in shopping:
    print(item)

# set comprehension <3
no_spam = [item for item in shopping if item != 'spam']
```

Dictionary

no duplicates + mapping

```
# Dictionary is iterable - unordered
# Dictionary is mutable

# create a dictionary - keep amount
shopping = dict('spam':1, 'flour':1, 'eggs':1)

# equivalent
shopping = {'spam', 'flour', 'spam', 'eggs'}

num_items = len(shopping)  # get size of set
print(shopping)            # print whole tuple
print(shopping[0])         # nope. dictionary is unordered
print(shopping['spam'])    # yes - this is how we use it
print('spam' in shopping ) # check f
shopping['spam'] = 3       # change value - yes it is mutable

# previous example also works - show reversed - these functions can be chained
for key, value in shopping.items():
    print('%s has value %s', (key, value))

# dict comprehension <3
no_spam = {key:value for key, value in shopping if key != 'spam'}
```

Example

this might feel weird

```
# well - let's see if you supposed
# all members must be of same type

# fast forward! Ignore this for now
class Stuff:
    # yeah i know - you're thinking underscores
    def __init__(self, name, is_weird):
        self.name = name
        self.is_weird = is_weird

    def __repr__(self):
      return '%s - %s' % (self.name, self.is_weird)

my_stuff = Stuff('hokus', True)
# applies for all - not only lists
wtf = ['foo', 5, 3.14, ('hmm..',), my_stuff]

for item in wtf:
    print item

# now is a good time to think
# why you use inheritance so much in java
```

Example

reading csv file

```
# don't need to do it like this - you can use pandas
# we are also not checking escape sequences etc
# demonstration of easyness

# create the file - this won't be necessary
with open('myfile.csv', 'w') as csv:
    csv.write('two,minutes,to,midnight\n')
    csv.write('run,to,the,hills\n')

# read the file we wrote
with open('myfile.csv', 'r') as csv:
    values = [each.rstrip().split(',') for each in csv]
print(values)
```

Ordered arguments

by order

```
# Function definitions
# None is like null
# returns something if specified else None
# arguments with order or named

def is_hot(temperature):
    hot = temperature >= 30
    return hot

def is_hot(temperature, threshold):
    hot = temperature >= threshold
    return hot

# default argument - must be after non default arguments
def is_hot(temperature, threshold=30):
    hot = temperature >= threshold
    return hot

# argument with order
print(is_hot(31))
# argument by name - doesn't need to be in order
print(is_hot(31, 30))
```

Keyword arguments

by keyword

```
# Function definitions
# None is like null
# returns something if specified else None
# arguments with order or named

def is_hot(temperature):
    hot = temperature >= 30
    return hot

def is_hot(temperature, threshold):
    hot = temperature >= threshold
    return hot

# default argument - must be after non default arguments
def is_hot(temperature, threshold=30):
    hot = temperature >= threshold
    return hot

# pass argument by keyword
print(is_hot(temperature=31))
# pass argument by keyword - doesn't need to be in order
print(is_hot(threshold=20, temperature=31))
```

Class - representation

hang on in there

```
# make class human readable!
# class definition

class Animal(object):

    # kinda java static
    has_soul = True
    # constructor
    def __init__(self, name):
        self.name = name
    def __repr__(self):
        return '\n'.join(['%s : %s' % (key, value)
                          for key, value in
                          self.__dict__.items()]

doug = Animal(name='Doug')
# this calls repr function
print(doug)
# all is going to be ok..
doug.kind = 'dog'
doug.age = 6
print(doug)
```

kwargs

In case someone wanted to try out kwargs

How to use kwargs

```
# I forgot to unpack the dictionary -_-
# example function that simply prints the arguments
def unknown_args(**kwargs):
    for key, val in kwargs.items():
        print('argument %s has value %s' % (key, val))

# create a dictionary of arguments
arguments = {'name': 'Ollie', 'age': 5, 'is_cool': True}
# call the function
unknown_args(**arguments) #  needs to be called with **
```

Don’t miss

language features

- *args and **kwargs in functions
- functions: zip, map, filter
- decorators
- class inheritance
- generators
- lambda expressions

libraries

- itertools, collections - python "stdlib"
- pandas - data structures
- numpy - scipy - number crunching
- pyyaml - yaml file format <- good for config files

The End

```
self.mute()
print('Any questions?')
```