Python Applied to Machine Learning and Statistics

Lecture 01: Python

Ricardo Cruz Pedro Costa Kelwin Fernandes

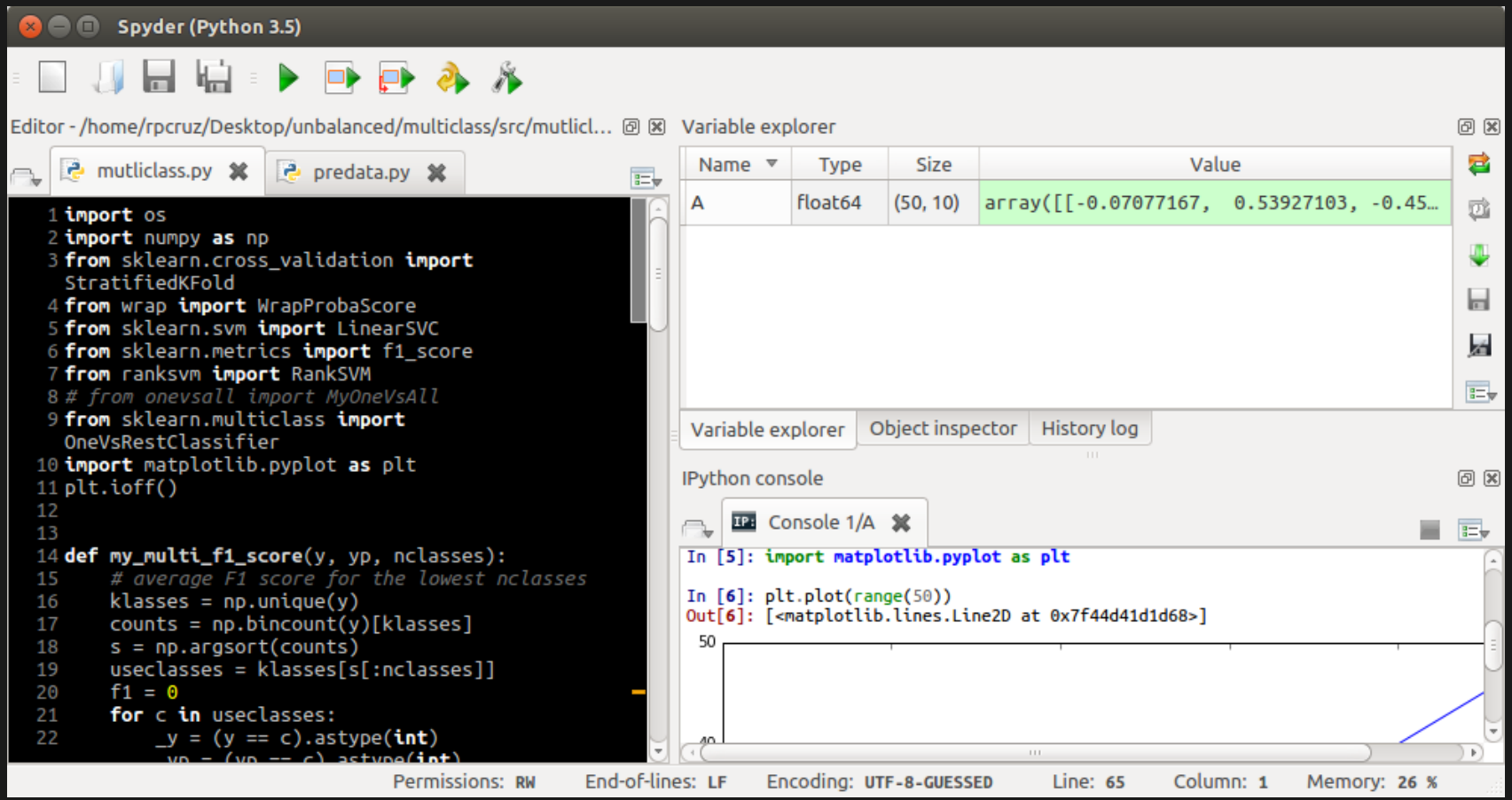
September 12, 2016

General Notes

- Python is a general purpose language
  - It is used in everything from user interfaces to web servers
  - The scientific computing community is strong, but is less centralized than Matlab
  - (But not as decentralized and messy as R :P)
  - In this first lecture, we will cover only language basics, not the packages for scientific computing

Editor

For these lectures, we are going to use **Spyder**. Spyder is an IDE[1] similar to the one offered by MATLAB by default.



[1] IDE = Integrated Development Environment

Syntax

Python

```python
def is_prime(n):
    for m in range(2, n):
        if n % m == 0:
            return False
    return True
```

Matlab

```matlab
function ret = is_prime(n)
    ret = true;
    for m=1:(n-1)
        if mod(n, m) == 0
            ret = false;
            break
        end
    end
end
```

Warning: In Python you **must** properly indent your code.

Exercise: create a function `fib(n)` which returns a list of the Fibonacci sequence up to `n`.

Create a list

```python
l = [1, 5, 2]
```

Access the list

```python
print(l[1])
# 5
```

Modify a list

```python
l[1] = 7
print(l[1])
# 7
```

Append element(s)

```python
l.append(8)
l += [9, 10]
```

Functions

In programming languages when calling a function, there are two possible behaviors:

- pass by value (or copy)
- pass by reference

Python uses what some people like to call ‘pass-by-object-reference‘. Everything is an object and the reference to the object is copied (not the object itself).

```
1 def fn(b):
2   print(id(b))
3   print(a is b)
4 a = []
5 print(id(a))
6 fn(a)
```

Output:

```
1 139663236727240
2 139663236727240
3 True
```

Functions

```
1 def fn(b):
2   b[0] += 1
3 a = [2]
4 fn(a)
5 print(a)
```

What is the value of a?

a = [3]

Functions

```
1 def fn(b):
2   b += [1]
3 a = [2]
4 fn(a)
5 print(a)
```

What is the value of a?

a = [2,1]

Functions

```
1 def fn(b):
2   b += 1
3 a = 2
4 fn(a)
5 print(a)
```

What is the value of a?

a = 2

Explanation: a is an immutable object (RO), therefore the operation '+=' modifies the reference, not the object itself.

```
1 v = 1
2 print(id(v))   # 10911712
3 v += 1
4 print(id(v))   # 10911744
```

Examples of immutable (RO) objects:

int, str, long, tuples

Examples of mutable (RW) objects:

lists, dictionaries

Loops

Python has some interesting functions that can help you in loops.

```
for n in range(10, 30, 2):
  print(n)
# 10 12 14 16 18 20 22 24 26 28

for i, n in enumerate(range(10, 20, 2)):
  print((i, n))
# (0, 10) (1, 12) (2, 14) (3, 16) (4, 18)

for a, b in zip(range(10, 20), range(40, 50)):
  print((a, b))
# (10, 40) (11, 41) (12, 42) (13, 43) (14, 44) (15, 45) (16, 46)
    (17, 47) (18, 48) (19, 49)
```

You will probably use these functions a lot in your loops: `range`, `enumerate` and `zip`.

Implementations

There are several implementations of the Python language:

- CPython
  - $\boxed{\text{source code}} \xrightarrow{\text{is compiled to}} \boxed{\text{bytecode}} \rightarrow$ execute
  - CPython is the most popular implementation of Python, and is developed by the Python Foundation
  - This is the implementation people *mean* when they say Python
- PyPy
  - $\boxed{\text{source code}} \rightarrow \boxed{\text{bytecode}} \rightarrow$ execute $\leftrightarrow \boxed{\text{machine code}}$
  - This is also a very popular and highly supported implementation
  - Faster; this is how Java and MATLAB work
- Jython
  - $\boxed{\text{source code}} \rightarrow \boxed{\text{Java bytecode}}$
  - The Java interpretor: $\boxed{\text{bytecode}} \rightarrow$ execute $\leftrightarrow \boxed{\text{machine code}}$
  - It outsources work to Java

CPython

While CPython does not interpret the source code directly (it first compiles to bytecode), it is still **much slower** than machine code.

Three ways to optimize it:

1. Cython
   - By defining types, CPython can directly compile to machine code:

```
def is_prime(int n):
  cdef int m
  for m in range(2, n):
    if n % m == 0:
      return True
  return False
```

   - We are not going to cover this
2. ctypes
   - Compile your C code into a shared object and then load it
3. Use python/package functions as much as possible to avoid cycles

List Comprehension

3. Use python/package functions as much as possible to avoid cycles

This includes such things as **vectorization**. We will cover vector/matrix functionality in the next lecture.

There are two ways to speed up list creation. Let us say we want to model $f(x) = 2^x$ from 0 to 50.

List comprehension

- List comprehension is considered the most Pythonic approach:

```
1 [2**x for x in range(50)]
```

[1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, ...]

To create a simple generator:

```
1 (2**x for x in range(50))
```

Lambda methods

- A set of methods based on functional programming:

```
1 map(lambda x: x**2, range(50))
```

[1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, ...]

Useful functions: `map`, `reduce`, `filter`.

Lambda

Usually list comprehension is considered the most Pythonic approach to speed up list generation.

But lambda methods, based on functional programming, are also interesting. In particular: `map`, `reduce` and `filter`.

They are known as lambda methods because they are often used with "lambda functions". These are anonymous functions, e.g. `lambda x: x*2`.

```
# exponential series of base 2
e = map(lambda x: 2**x, range(10))
e = list(e)  # instantiate as a list (python3)
print(e)  # [1, 2, 4, 8, 16, 32, 64, 128, 256, 512]
print(reduce(lambda a,b: a+b, e))  # 1023  (sum numbers)
print(list(filter(lambda n: n < 50, e)))  # [1, 2, 4, 8, 16, 32]
```

Exercise: Filter **even numbers** of your `fib` using `filter()` and using list comprehension.

Infinite Generators

Memory is expensive, so in scientific computing, it is sometimes cheaper to read a file line-by-line or do calculations number-by-number.

Exponential series:

```
def exp(base):
  i = 0
  while True:
    yield base**i
    i += 1
```

Using:

```
series = exp(2)  # exponential series of base 2
print(next(series))
# 1
print(next(series))
# 2
print(next(series), next(series), next(series), next(series))
# 4 8 16 32
print([next(series) for i in range(10)])
# [64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384, 32768]
```

Exercise: create a Fibonacci series until infinity!

Copy

```
1 from copy import copy
2 a = [1, 2, 3]
3 b = copy(a)
4 a[0] = 5
5 print(b)
```

What is the value of `b`?

`b = [1, 2, 3]`

We have copied the object.

Copy

```
from copy import copy
a = [[1, 2, 3], [8, 9, 10]]
b = copy(a)
a[0][0] = 5
print(b)
```

What is the value of `b`?

`b = [[5, 2, 3], [8, 9, 10]]`

Explanation: We have copied the object, which included all object references it contains. But we did not copy the objects inside the object.

We need deep copy for that.

```
from copy import deepcopy
a = [[1, 2, 3], [8, 9, 10]]
b = deepcopy(a)
a[0][0] = 5
print(b)  # [[1, 2, 3], [8, 9, 10]]
```

Dictionaries

Python supports the "dictionary" structure. They are also known as: "hashes" or "maps" in other languages/libraries.

For example:

```
d = {'key1': 5, 'key2': 10}
print(d['key1'])  # 5
d['key3'] = 30    # adding a key
```

You can also use **dictionary comprehension.**

```
d = {'key%d' % i: i*2 for i in range(5)}
print(d)
# {'key0': 0, 'key1': 2, 'key4': 8, 'key2': 4, 'key3': 6}
```

Starred expressions

Unpacking a list:

```
def fn(a, b, c):
    print(a, b, c)

fn(*[1 2 3])
# 1 2 3
```

Unpacking a dictionary:

```
def fn(a, b, c):
    print(a, b, c)

fn(**{'c': 3, 'a': 1, 'b': 2})
# 1 2 3
```

Same syntax for functions:

```
def func(*args, **kw):
    # args now holds positional arguments, kw keyword arguments
```

Python Precision

What is number precision in Python?

32-bits? 64-bits? 128-bits?

Actually, Python uses arbitrary-precision arithmetic for integers (like Lisp and Haskell). Your number can be as big as you want without using any extra package.

As we will see in the next lecture though, we will probably want to define the precision of our numbers when doing scientific computing to better control memory usage.

Working with Strings

In Python, it is much easier to work with strings than MATLAB.

In MATLAB, they are vectors. In Python, they have their own type, and operator semantics are overloaded for them:

To concatenate strings: `print('Hello' + ' ' + 'World')`

To format strings, I like C-style `printf` notation. In Python, the % operator is available for that. For instance, let us say we want to print several famous constants:

```
1 import math
2 C = (('pi', math.pi), ('e', math.e))
3 for name, value in C:
4   print('%-10s %.4f' % (name, value))
```

Exercise: print the first 18 Fibonacci numbers in a line as 4-padded integers like this: `0001, 0001, 0002, 0003, 0005, 0008, 0013, 0021, 0034, 0055, 0089, 0144, 0233, 0377, 0610, 0987, 1597, 2584`

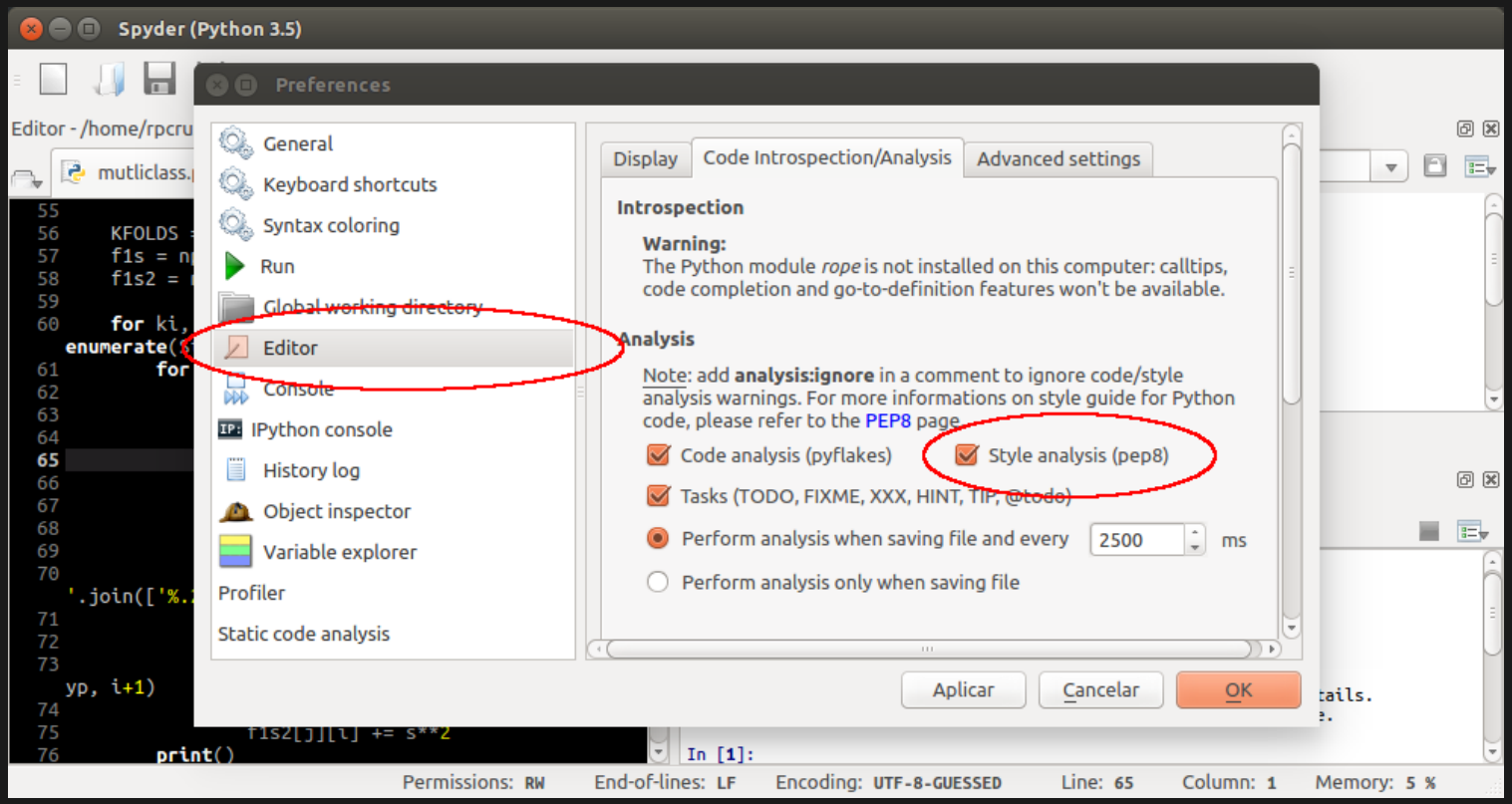
PEP8

Using a pretty style makes reading your code much easier.

In Python, there is a coding style that is recommended called PEP8.

Exercise: Enable code style checking in Spyder and fix the errors.

Under the Preferences window:



Classes

```
1 class Animal:
2   def talk(self):
3     raise NotImplementedError('Implement me')
4
5 class Cat(Animal):
6   def talk(self):
7     print("Miau")
8
9 class Human(Animal):
10  def talk(self):
11    print("Bla bla")
12
13 Cat().talk()  # what is the output?
```

Two notes:

- Python is very dynamic: there are no formal contracts like abstract methods
- in Python, the reference to the object itself is passed explicitly

Exercise: Embed your `fib()` function inside a `Fib.next()` method within a `Fib` class.

Design Patterns

Not Python-related, but important to point out. In computer engineering, there are some important design patterns when working with classes:[2]

Wrapper: when placing a class inside another, possibly changing its interface

Decorator: class that extends the functionality of another while keeping the same interface

Proxy: a class that acts as an intermediary for something else

Iterator: a class to be used within a cycle (also known as generator in python)

Factory: a class that creates objects of different types abstractly for the user

Among other...

[2] `https://en.wikipedia.org/wiki/Software_design_pattern`

Modules

Different ways to access a module:

```
import fib
fib.whatever()
```

```
from fib import whatever
whatever()
```

```
from fib import *
whatever()
```

(import everything into your namespace — usually not recommended)

Inside each directory, you **must** have a `__init__.py` file where you can put initialization code (or just leave it empty).

Exercise: Move your `Fib` class to its own file inside a `fib` subdirectory.

Save Session

There are two ways to save your session/variables to a file:

- `pickle`: binary file (python specific)
- `json`: text file (universal)

Pickle is the native Python approach:

```
import pickle
pickle.dump(d, open('save.p', 'wb'))
# and then you can just load it again:
d = pickle.load(open('save.p', 'rb'))
```

Save Session

Another approach which is more universal and whose files are readable is **json.**

```
import json
json.dump(d, open('save.json', 'w'))
# and then you can just load it again:
d = json.load(open('save.p', 'r'))
```

json files can be read by an editor:

```
{"key0": 0, "key1": 2, "key4": 8, "key2": 4, "key3": 6}
```

BUT json files are **bigger** than pickle files. A rule of thumb is to save opaque objects as pickle, and simpler variables as json.

We will look at the CSV format (for tabular data) in the next lecture.

Memory Management

Unlike C where you need to use extra packages for memory management – or do it yourself – Python has native automatic memory management.

```
a = Cat()     # a Cat instance is allocated (Cat.refcount=1)
b = a         # Cat.refcount=2
a = Human()   # Cat.refcount=1
b = 0         # Cat.refcount=0 => Cat instance eliminated
```

- Java and R use a background process (called the **garbage collector**) to remove inaccessible memory (this process runs from time to time)
- CPython removes memory immediately by a **reference counting** system, but also does occasional garbage collection to find circular references

Circular references: CPython's normal reference counting does not catch cases like this:

```
a = Cat()  # Cat.refcount=1
a.ptr = a  # Cat.refcount=2
a = 0      # Cat.refcount=1
```

Parallelism

In computing, parallelism is usually achieved by:

- Running various processes
- Running various threads within the same process

There are of course trade-offs when choosing the parallelism approach.

Threads share resources with each other, since they belong to the same process.

Each **process** has its own memory and must communicate through sockets or other means.

In CPython, **multiprocessing** is the favorite approach to parallelism — the reason is that in CPython memory management is not thread-safe. A mutex called the global interpreter lock (GIL) prevents multiple native threads from executing CPython bytecodes at once.

Multiprocessing

Multiprocessing can be done easily in Python through the `multiprocessing` package. Example:

```
def avg_file(file):
  import numpy as np
  return np.mean(np.load_txt(file))

import multiprocessing
cpus = multiprocessing.cpu_count()
p = multiprocessing.Pool(cpus)
files = os.listdir('data')
```

`multiprocessing.Pool` creates a pool of worker processes. Then you can easily distribute tasks by using one of the following:

```
print(p.map(avg_file, files))
```

```
for f in files:
  p.apply_async(avg_file, f)
for result in p.get():
  print(result)
```

`Pool.apply()`	`Pool.apply_async()`	one argument
`Pool.map()`	`Pool.map_async()`	map arguments

Multiprocessing

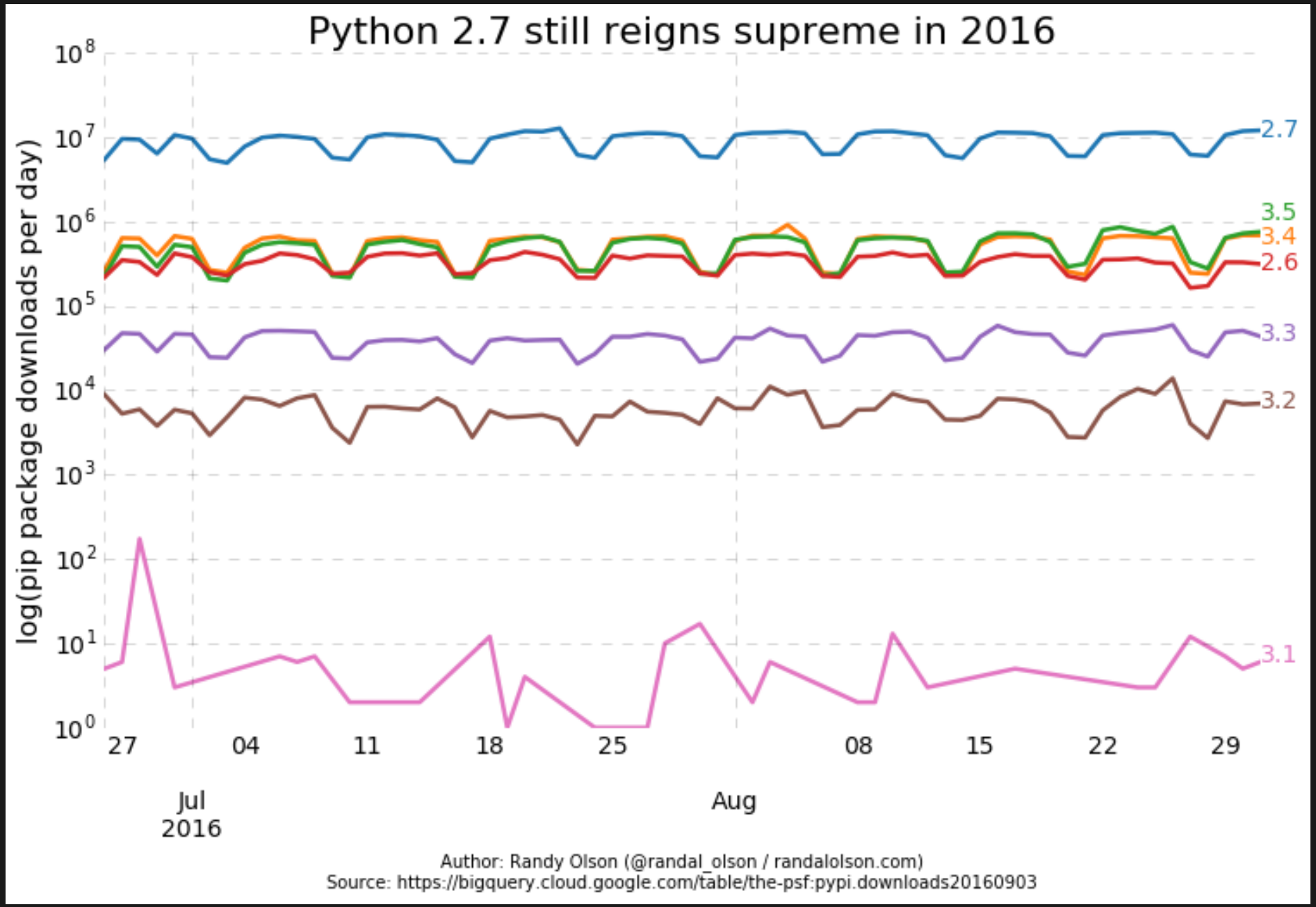
Exercise: Use `multiprocessing` to calculate the first 20 Fibonacci numbers by initializing the series with i, j with $i, j \in \mathbb{I}$ and $i \leq j$ (use `random.randint(a, b)` to sample numbers).

Reference:

`Pool.apply()`	`Pool.apply_async()`	one argument
`Pool.map()`	`Pool.map_async()`	map arguments

Note for Windows users:

Windows does not implement the POSIX `fork()` method and is generally not friendly to `multiprocessing`. You should guard the parent code with `if __name__ == '__main__'` to avoid it being executed.


Python 2.7 still reigns supreme in 2016
log(pip package downloads per day)
10^8
10^7
10^6
10^5
10^4
10^3
10^2
10^1
10^0
2.7
3.5
3.4
2.6
3.3
3.2
3.1
27
04
11
18
25
08
15
22
29
Jul
2016
Aug
Author: Randy Olson (@randal_olson / randalolson.com)
Source: https://bigquery.cloud.google.com/table/the-psf:pypi.downloads20160903

Python2 vs Python3

Python2 will be discontinued in 2020. The major differences are that:

- In Python3, functions like `range()` are generators/iterators, while in Python2 they are lists. In most use cases, this will make no difference to you.
- In Python3, strings are codified in Unicode by default. This makes it nicer to work with when using, for instance, Portuguese characters like ç or á.
- Probably the when you'll notice the most is that in Python2 `print 'Hello World'` was valid. In Python3, you need to use parenthesis, `print('Hello World')`
- There are many other small ones. **Most importantly, not all packages are yet compatible with Python3.** (Though 99% are.)

Homework

Several Python exercises are available at:

`https://www.hackerrank.com/domains/python/py-introduction`

Please do one exercise of each module for the next class (except for numpy).