Course Authored By:
###
###
K.Srinivasan
NeuronRain Documentation and Licensing: http://neuronrain-documentation.readthedocs.io/en/latest/
Personal website(research): https://sites.google.com/site/kuja27/

This is a non-linearly organized, code puzzles oriented, continually updated set of course notes on Python language. This complements NeuronRain course materials on Linux Kernel, Cloud, BigData Analytics and Machine Learning and covers fundamentals of Python.

----
6 February 2017

Code Reference:
----
https://github.com/shrinivaasanka/asfer-github-code/blob/master/python-src/CNFSATSolver.py

Python's object oriented paradigm is quite similar to most of the languages like Java and C++ but the difference in number of lines code for same algorithm between Python and C++/Java is what makes it favourable for Natural Language Processing and writing microservices (services with small functionality which are interconnected). Python is more Haskell or LISP functional language-like and achieves both worlds of functional programming and imperative procedural programming by Lambda on-th-fly functions. Previous code demonstrates some of the minimum basic features of Python:

- Python Classes
- Tuples
- Lists
- List Slicing
- Set operations on python objects
- Control structures (for, if)
- Python object member functions and self keyword

Python classes are defined with class <class>(<baseclass>) and member functions are defined with def <function>(). Base class by default is object unless explicitly stated. Tuples are ordered pairs of arbitrary dimensions equivalent to vectorspaces in mathematics defined with (). Lists are equivalent to arrays in C defined with [] subscripts. Accessing an element in both tuples and lists are by [] subscript. Concepts of slicing is central to list comprehension in Python. Slicing can return a contiguous subset of a list by [<start>:<end>] notation. When either start or end is ignored implicit list start and end are assumed. For loops in python can iterate over any "iterable" object. Iterables include lists, dictionaries and user defined containers. if..else..elif is the python equivalent of conditional clauses with truth values being boolean keywords "True" or "False" which are builtins. Python classes denote self keyword to be the present object instantiated (equivalent to "this" in C++)

----

27 February 2017 - Python Generators and Yield

Python has a notion of iterables where any sequential data structure can be made to return an element and resume from where it left to return the next element in the sequence. This is quite useful for problems which need to remember the last element accessed and resume from next element (can be constrasted with static keyword in C/C++ which live across function invocations with global scope) - typical streaming scenario.

Streaming Abstract Generator implemented in :
https://github.com/shrinivaasanka/asfer-github-code/blob/master/python-src/Streaming_AbstractGenerator.py
is based on this idiom. It basically is facade frontend abstraction for multiple backend streaming datasources. It overrides the __iter__() method to access an element and yield it instead of returning it. User of this class, instantiates with desired data storage constructor arguments and iterates through it without any knowledge of how the data is accessed. In python terms, data is "generated" and "yielded" iteratively in a loop accessing consecutive elements. No storage is allocated by generator explicitly and backend client objects for HBase/Cassandra/Spark/Hive/File datasources handle them internally.

This is a typical example of Iterator/Facade design pattern listed in Gof4.

----

844. (THEORY and FEATURE) 6 January 2018,18 July 2020 - Currying and Partial Function Application in Python - related to all sections on Recursive Lambda Function Growth algorithm implementation for learning lambda functions from Natural Language Texts and Text analytics

----

Python has functional programming (Haskell) equivalents of Currying and Partial Application support. Currying converts a function of n parameters to n functions of one parameter each invoked in nesting, and returns a function in each function. Partial Application Function is similar to currying but takes 2 arguments instead of 1 in currying, and returns a function. Code examples for these are below - committed to code/Currying.py and code/PartialFunctionApplication.py

----

Currying.py:

===========

```
Curried :
===========
6 5 4 3 2 1

------------------------------
PartialFunctionApplication.py:
------------------------------
=========================================
Equivalent Partially applied functions
=========================================
1 2 3 4 5 6
1 2 3 4 5 6
1 2 3 4 5 6
1 2 3 4 5 6
1 2 3 4 5 6
1 2 3 4 5 6
```

Currying by Partial Functions (as Beta reduction) are used in Recursive Lambda Function Growth - Graph Tensor Neuron Network algorithm implementation of NeuronRain which converts a natural language text to tree composition of functions of 2 arguments (a function operator acting on two operands) by traversing a random walk or cycle of the textgraph.
Example:

```
maximum_per_random_walk_graph_tensor_neuron_network_intrinsic_merit=
('one((integer(digit,(definite_quantity(abstraction,(measure(number,(comm
and(act,(speech_act(psychological_feature,(order(abstraction,(event(order
,(speech_act(event,(act(abstraction,(digit(three,(psychological_feature(n
umber,(abstraction(measure,(definite_quantity(integer,command))))))))))))
)))))))))))))))', 10.804699467199468)
```

----

7 August 2018 - Python Dictionaries

Python dictionaries are the hashtable implementations using linear probing(open addressing) to find next available slot (preferred to Chaining/Buckets). Open addressing in python applies the following function recurrently to find next available slot:

```
x = 5*x + perturb + 1
perturb >> PERTURB_SHIFT (some constant)
```

This function has been found to be optimal in python benchmarks. Simulating buckets/chaining in builtin dictionaries is therefore done by an alternative dictionary implementation defaultdict() which initializes the dictionary to a default key whose type is defined by argument to constructor. An example, chaining/buckets has been described in code/Dictionaries.py which prints :

```
['__class__', '__cmp__', '__contains__', '__delattr__', '__delitem__',
'__doc__', '__eq__', '__format__', '__ge__', '__getattribute__',
'__getitem__', '__gt__', '__hash__', '__init__', '__iter__', '__le__',
'__len__', '__lt__', '__ne__', '__new__', '__reduce__', '__reduce_ex__',
'__repr__', '__setattr__', '__setitem__', '__sizeof__', '__str__',
'__subclasshook__', 'clear', 'copy', 'fromkeys', 'get', 'has_key',
'items', 'iteritems', 'iterkeys', 'itervalues', 'keys', 'pop', 'popitem',
'setdefault', 'update', 'values', 'viewitems', 'viewkeys', 'viewvalues']
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81}
```

defaultdict(<type 'list'>, {0: [0], 1: [1, 81], 4: [4, 64], 5: [25], 6: [16, 36], 9: [9, 49]})

References:
----
1. CPython PyDictObject source - code comments - https://github.com/python/cpython/blob/master/Objects/dictobject.c
2. Beautiful Code - [Greg wilson]

----
7 October 2018 - Python Reflection, Derived Classes and Class Methods
----

Python has support for reflection in the form of function and code objects. Python function objects are accessed by __func__ member of any function considered as an object. Python classmethods are decorators declared before defining a function by @classmethod annotation and defined as class method and defined by def <func>(cls, ...) declarative containing cls keyword. Class methods are useful for defining a classwide method common to all instances . This is different from static keyword in other languages and there is a @staticmethod decorator in python, the static equivalent. Class methods called on derived class take derived class objects for cls. An example for reflection and class methods is in code/Reflection.py and logs in code/logs/Reflection.log.7October2018

----
13 March 2019 - Async/Await - Asynchronous IO in Python 3.7

Python 3.7 supports asynchronous invocations of coroutines by async and await pairs of keywords similar to promise-future in C++. An example Chatbot Client and Server in AsyncIO_Client.py and AsyncIO_Server.py define two classes for Chat Server and Client which implement connection_made() and data_received() interface functions. Notable feature in 3.7 is the "async def" for function declarations in lieu of @asyncio coroutine decorators in previous versions of Python 3. Both client and server instantiate a loop object and invoke create_connection() and create_server() respectively for client and server by await semantics. Notion is to make client-server transport completely event driven by await and non-blocking by async.This example applies as a pattern to any algorithm doing asyncio. Logs in logs/AsyncIO_ClientServer.log.13March2019 show a sample chat.

References:

1.Python 3.7 documentation - Async/Await - https://docs.python.org/3/library/asyncio-protocol.html

----
14 November 2019, 23 April 2020 - Multidimensional Array Slicing - List comprehension, slice() function and NumPy - and StringIO

Python has in-built slice() function which is an iterator for array indices to slice. Numeric Python

NumPy has library support for multidimensional array slicing by subscripts. Code example code/Slicing.py demonstrates 5 variants of slicing of a 2 dimensional ndarray parsed from a string text. Text of multiple lines separated by newline "\n" is read by StringIO object by genfromtxt() and parsed to a multidimensional array by delimiter ",". Five different slicings in the example are:

1. slicedarray1 - equal slices for 2 dimensions by slice()
2. slicedarray2 - unequal slices for 2 dimensions by slice() - only larger slice is effected
3. slicedarray3 - slice() for one dimension and tuple of indices for the other - only indices from tuple are effected
4. slicedarray4 - equal slices of 2 dimensions with an added step parameter - only one dimension is sliced
5. tuple of indices for both the dimensions raises Python error:

```
Traceback (most recent call last):
  File "Slicing.py", line 23, in <module>
      slicedarray3=parsedarray[(0,3),(0,3)]
IndexError: index 3 is out of bounds for axis 0 with size 2
```

Code example chooses either Slicing or List comprehension to extract a slice by a flag - List comprehension being the most fundamental Python primitive is the obvious choice for slicing. Following list comprehension:

```
slicedarray5=[row[2:5] for row in parsedarray]
```

extracts a rectangular slice:

```
[['1', '2', '3', '4', '5', '6', '7', '8'], ['9', '10', '11', '12', '13', '14', '15', '16']]
slicedarray5 - list comprehension:
[['3', '4', '5'], ['11', '12', '13']]
```

----

14 December 2019 - Rounding off, Floating point division, Python 2.7 and Python 3.7.5, PDB

Following code snippet defined within a class of code/RoundOff.py demonstrates difference in division behaviour between Python 2.7 and Python 3.7.5 by disassembly of Python bytecode in PDB debugger:

```
x1 = (1-2)/2
  x2 = (2-1)/2
```

```
=================================================================================
Python 2.7
=================================================================================
# python RoundOff.py
('x1:', -1)
('x2:', 0)
=================================================================================
Python 3.7.5
=================================================================================
# python3.7m RoundOff.py
x1: -0.5
x2: 0.5
```

Python 3.7.5 does auto type promotion to float while Python 2.7 rounds off to floor. PDB is imported by -m pdb in commandline and disassembler is imported as dis in pdb. Roundoff class is loaded in PDB. PDB Python VM bytecode disassembly shows absence of binary divide opcode in Python 3.7.5

----
26 June 2020 - Object Marshalling and Unmarshalling - Python Serialization and Persistence
----
Python has variety of infrastructure to support serialization and deserialization of objects - some of
them being Pickle,Marshal,DBM and Shelve modules. While pickling is widely used, marshal,dbm and shelve are lowlevel alternatives - marshal is most widely used for persisting compiled binaries (.pyc) and restricted to python types while shelve and DBM depend on file and linux database backends and can persist arbitrary user defined types as persistence dictionaries of name-values. Thus lookup is easier making them powerful than pickle and marshal. Code example Marshal.py demonstrates the serialization and deserialization of an example Python 3.7 class and primitive list datatypes. Class Marshal defines __init__(), marshal(),unmarshal() and sync() functions which respectively wrap following shelve innards:

- init - opening a file persistence by shelve.open() in writeback=True mode and retrieve serializer dict
- marshal - populating serializer dict by object name ("array1" and "exampleobject1") and object.
- unmarshal - retrieval of a persisted object by object name from serializer dict
- sync - which synchronizes the persistence if writeback=True

Marshal databases and logs are in:
code/MarshalDB.bak
code/MarshalDB.dat
code/MarshalDB.dir
code/logs/Marshal.log.26June2020

Fields of deserialized Example() object and array are printed by print():

Marshalling object: array1
Shelve serializer: [('array1', [1, 2, 3]), ('exampleobject1', <__main__.Example object at 0xb7396aec>)]
Unmarshalling object: array1
Shelve serializer: [('array1', [1, 2, 3]), ('exampleobject1', <__main__.Example object at 0xb7396aec>)]
[1, 2, 3]
Marshalling object: exampleobject1
Shelve serializer: [('array1', [1, 2, 3]), ('exampleobject1', <__main__.Example object at 0xb7396aec>)]
Unmarshalling object: exampleobject1
Shelve serializer: [('array1', [1, 2, 3]), ('exampleobject1', <__main__.Example object at 0xb746188c>)]
field1: field1
field2: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

----

30 July 2020 - Pointers, C-types, Copy-On-Write, Mutables, Immutables in Python 3.7.5

Python compiles to bytecodes which are executed on Python Virtual Machine. Basic datatype objects are
immutable while iterables are mutable in python. Code example Pointers.py defines a function to
simulate pointers on Python by importing ctypes C wrapper library and instantiating pointers
to some string and integer objects by c_wchar_p() and c_void_p(). Illegal access Exception while assigning to immutable string literal is handled and pointer to string is created by writing a copy (Copy-On-Write) of original string which is different from C pointers. Similarly pointer copy to an integer data is assigned to while leaving the original unchanged. An alternative to C-type string creation by create_string_buffer() is demonstrated.

```
Logs from code/logs/Pointers.log.30July2020 :
Immutable - sentence1: This is a sentence
Exception: 'str' object does not support item assignment
Pointer to sentence1 string: c_wchar_p(3075052112)
Changed pointer copy of sentence1: This is different sentence
sentence2 - RAM contents after create_string_buffer():
b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00'
Pointer to integer1: c_void_p(200)
immutable - integer1: 100
Changed pointer copy of integer1: 200
```

References:
----

1. Python 3.8.5 documentation -
https://docs.python.org/3/library/ctypes.html

----

862. (THEORY and FEATURE) NeuronRainApps - NeuronRain Usecases - MAVSDK Python Async I/O Drone Simulator - related to 637,713,722 and all sections on Drone Autonomous Delivery, Drone Electronic Voting Machines, Large Scale Visuals/Urban Sprawl/GIS Analytics, VIRGO PXRC Flight Controller Kernel Driver - 18 August 2020

----

1. This commit implements Drone_MAVSDK_Client.py and Drone_MAVSDK_Server.py asynchronous Drone I/O by importing MAVSDK (Micro Air Vehicle SDK) Python library which contains an inferface for drone telemetry - mavsdk_server. Earlier NeuronRainApps usecase implementations for Autonomous Online Shopping Delivery (and Conceptual Set Partition Drone Electronic Voting Machines) depend on Dronecode Python SDK which might alternatively import MAVSDK.
2. Asynchronous I/O Chatbot Python class example has been augmented to import mavsdk and defines a new asynchronous function drone_async_io() which takes a Drone System object, Drone port, Drone action as arguments and accordingly async-invokes (await) MAVSDK internal functions for those actions.
3. Because of absence of a licensed Drone, this is a conceptual-only usecase implementation of a Drone simulator and "No System" exceptions are thrown which are handled and printed. Standalone product-specific Simulators (e.g jMAVSIM, Gazebo and AirSim for PX4 -
https://dev.px4.io/v1.9.0/en/simulation/jmavsim.html, ROS-Simulator -

https://dev.px4.io/v1.9.0/en/simulation/ros_interface.html) could be used for development testing.
4. Drone action clauses for "info","takeoff","arm","camera","goto_location" have been implemented in drone_async_io() which respectively print flight information, control takeoff, arm the AV, control camera video streaming, and pass on Longitude-Latitude-Altitude-Yaw GPS destination data for autopilot of the AV.
5. drone_async_io() is an async member of DroneMAVSDKClient class. Drone object is instantiated by System() and drone_async_io() initializes connect() to mavsdk_server Drone server port.

----

863. (THEORY and FEATURE) MAVSDK Python jMAVSIM Drone flight simulator - Proof-of-Concept implementation related to 862 and all sections on Drone Autonomous Delivery, Drone EVMs, Large Scale Visuals/Urban Sprawl/GIS Analytics, VIRGO PXRC Flight controller kernel driver - 26 August 2020

----

1. This commit revamps Drone_MAVSDK_Client.py earlier to define a new command "all" in drone_async_io() which connects to a drone, prints its flight information, arms, takes-off, navigates, streams visuals to ground station and lands a drone - everything in a simulator (jMAVSIM)
2. Before arming the drone its connection state, UUID and GPS health checks are preferable code for which has been introduced in 2 async iterators. "If" clause for Command "all" is a ubiquitous pattern which might be frequently necessary in Drone Autonomous Delivery and Drone electronic voting machines.
3. Drone_MAVSDK_Client.py connects to a PX4 jMAVSIM flight simulator which can be installed by instructions in https://dev.px4.io/v1.9.0/en/setup/dev_env_linux.html (Section on JMAVSIM and Gazebo simulation) and https://dev.px4.io/v1.9.0/en/simulation/jmavsim.html. PX4 Firmware, ROS and ECL dependencies are installed by sourcing shell script https://raw.githubusercontent.com/PX4/Devguide/v1.9.0/build_scripts/ubuntu_sim.sh which clones PX4 dependencies and creates build scripts. PX4 jMAVSIM SITL (Software in the loop) is built by target "make --debug px4_sitl jmavsim" in PX4/Firmware source which starts the Simulator GUI and SITL CLI pxh> prompt.
4. Drone_MAVSDK_Client.py is either executed from apython (asynchronous python CLI installed by aioconsole) or usual python commandline interface which connects to jMAVSIM GUI server.
5. jMAVSIM flight simulation logs for a multirotor aerial verhicle is at code/logs/PX4_Drone_JMAVSIM_Flight_Simulation.log.26August2020 which shows the jMAVSIM server side logs for GPS, arm, vertical takeoff, navigation and vertical landing. Camera streaming is failed with Error "DENIED".
6. drone_async_io() might be significantly augmented and imported across every Drone dependent analytics code in NeuronRain which can be developed and tested on a 3D Virtual Reality simulation without a real drone.
7. Efficient Drone Autonomous Delivery is an NP-complete Travelling Salesman-Hamiltonian Cycle problem which traverses lowest cost circuit connecting set of delivery points on a transporation network graph.
8. Drone FOSS Code Repositories and References:
    8.1 MAVSDK Python - https://github.com/mavlink/MAVSDK-Python
    8.2 PX4 Drone Autopilot Firmware - https://github.com/PX4/Firmware
    8.3 PX4 Estimation and Control - https://github.com/PX4/ecl
    8.4 PX4 JMAVSIM - https://github.com/PX4/jMAVSim

8.5 Auterion MAVSDK Java - Android Client - QGroundControl - https://auterion.com/getting-started-with-mavsdk-java/

8.6 PX4-ROS-Gazebo simulator - Graphic Illustration - MAVROS - https://dev.px4.io/v1.9.0/en/simulation/ros_interface.html

8.7 PX4: A node-based multithreaded open source robotics framework for deeply embedded platforms
 - "...Our system architecture is centered around a publish-subscribe object request broker on top of a POSIX application programming interface. This allows to reuse common Unix knowledge and experience, including a bash-like shell. We demonstrate with a vertical takeoff and landing (VTOL) use case that the system modularity is well suited for novel and experimental vehicle platforms. We also show how the system architecture allows a direct interface to ROS and ..." - https://ieeexplore.ieee.org/document/7140074

9. jMAVSIM replay logs have been committed to code/logs/jMAVSIM_replay_logs/

----

5 September 2020,7 September 2020 - Python Decorators, Static Type Checking, Type Hints, Mypy and IDE typecheckers, Final constant type hint, final decorator in Python 3.8

Python traditionally implements Duck typing or Dynamic typing and types of objects could be checked
at runtime by type() and isinstance(). Decorators in Python are features which could be used to instrument another function and return a wrapped new function having additional code. Lack of support for static
type checking and constant definitions in Python have been answered in Python 3.8 which implements series
of PEPs for final decorator for final methods which cannot be overridden in derived classes and Final
type hint which aids Typecheckers (mypy, PyCharm IDE) to statically analyze code and flag type errors.
Code example Decorators.py defines two classes - Base and Derived - which import various typecheck artefacts from typing module. Class BaseDecorated defines a constant var1 by type hint Final and a method cannotoverride() by @final decorator which can be type-enforced by linters in mypy (http://mypy-lang.org/) and PyCharm. Both base and derived classes define member functions to which argument type hints and return type hints are annotated by ":" and "->" operators. Most importantly member function funcdecorator() defines an inner function wrapper() which takes a function argument and instruments additional code around it and returns a new decorated function by typing.cast() typecasting. Declaration of T defines a Callable type variable (TypeVar) of arbitrary number of arguments specified by Any. Logs code/logs/Decorators.log.5September2020 capture the decorator callstack.

----

866. (THEORY and FEATURE) A* (A-Star) Best First Search Algorithm Implementation - Python 3.8.5 - 8 September 2020, 27 October 2020 - related to all sections on Drone Autonomous Delivery Navigation, Drone Obstacle Avoidance, Drone Electronic Voting Machines and Graph Analytics

----

1.This commit implements the standard A-Star Path Finding algorithm widely used in Robotics as a general

purpose graph search implementation which could be invoked by multitude of NeuronRain code dependent on Graph Analytics as well as a routine prerequisite for motion planning in NeuronRain Drone Navigation.
2.A-Star algorithm of [Hart-Nilsson-Raphael] described in https://en.wikipedia.org/wiki/A*_search_algorithm is the reference for this implementation in Python 3.8.5.
3.This implementation uses a plain list in place of a Priority Queue for Open set (Discovered nodes)
4.A-Star algorithm improves upon Dijsktra's Shortest Path by finding the path which minimizes the cost function with the help of a heuristic:

argmin(f(n) = g(n) + heuristic(n))

where f(n) is the fscore map, g(n) is the cost of traversing to node n (gscore map) from start and heuristic(n) is the estimated cost of traversing to end vertex from n.
5.Logs in code/logs/AStar_BestFirstSearch.log.8September2020 compute the Best First Search Path [3,6,7] from node 3 to node 7 in an 8 vertex graph denoted by adjacency matrix which marks lack of edges by -1.
6.Navigation in Drone Autonomous Delivery and EVMs is a TSP-Hamiltonian Cycle NP-complete problem wherein Drone has to efficiently visit set of all delivery points-voter residences once while A-Star motion planning is necessary for finding the least cost trajectory between longitude-latitude-altitude of any two delivery points or voter residences (by looking up the addresses in a map service e.g Google Maps).

References:

866.1 Finding Closest Pair of Points - O(NlogN) divide and conquer algorithm better than naive O(N*N) bruteforce for obstacle avoidance - Section 35.4 - Chapter 35: Computational Geometry - Algorithms - [Cormen-Leiserson-Rivest-Stein] - Page 908 - "... System for controlling air or sea traffic might need to know which are the two closest vehicles in order to detect potential collisions ..." - quite relevant for Drone swarms

----

871. (THEORY and FEATURE) Generating all possible permuted strings of an alphabet - related to 843 and all sections on Social networks, Bipartite and General Graph Maximum Matching, Symmetric Group, Permanent, Boolean majority, Ramsey coloring - Number of Perfect (Mis)Matchings - 28 October 2020

1.Related to a question in IIT-JEE: Find sum of integers > 10000 having only 0,2,4,6,8 as digits without repetition (implies < 99999 because more digits would repeat).
2.This commit implements Permutations.py utility which generates all possible permutations of a list of integers and creates permuted integers from them. Generating permutations is required in section 843 for perfect (mis)matches.
3.In the function genpermutations() random permutation is obtained from numpy.random.permutation() and its numeric equivalent is hashed to a dictionary iteratively till all permutations are generated (which is N!).
4.Finding sum of the permuted integers is tricky and makes use of the fact that every digit in the list of permutation is independently and identically distributed - For example, there are 120 (5!) integer permutations of 0,2,4,6,8 and each of [0,2,4,6,8] occurs 120/5 = 24 times per digit in the permutations.
5.Thus per digit sum is 24*8 + 24*6 + 24*4 + 24*2 + 24*0 = 480 and following schematic depicts carry forward:

```
53+ 53+ 52+ 48+
|  |  |  |  |
....(480 for each digit)
|  |  |  |  |
-------------
 533 3  2  8  0
```

and sum of the permuted integers is 5333280.
6.Permutations.py demonstrates this fact and prints:

```
('summation:', 5333280)
('permutations:', defaultdict(<type 'int'>, {46082: 56, 24068: 97, 20486:
24, 86024: 36, 82604: 75, 60428: 15, 62480: 52, 4628: 81, 24086: 70,
86042: 65, 64028: 40, 24608: 85, 80426: 43, 8246: 107, 26804: 95, 84026:
35, 60482: 55, 8264: 41, 4682: 78, 80462: 28, 64082: 80, 42068: 9, 24860:
91, 2648: 110, 84062: 39, 62048: 99, 42086: 21, 24680: 100, 68204: 77,
42608: 71, 6248: 116, 84602: 86, 2684: 108, 82046: 82, 46208: 51, 48260:
44, 84620: 7, 68240: 57, 20648: 79, 82460: 58, 4268: 61, 40628: 8, 42680:
2, 6842: 62, 86204: 83, 4286: 68, 46280: 109, 20684: 102, 82640: 30,
46802: 69, 26840: 113, 4826: 120, 86240: 6, 64208: 31, 46820: 76, 24806:
14, 8426: 63, 84206: 13, 80624: 3, 28406: 18, 4862: 98, 80642: 72, 8462:
96, 64280: 32, 62084: 48, 6428: 49, 2846: 89, 64802: 92, 84260: 27,
26408: 22, 28460: 26, 2864: 12, 68402: 104, 64820: 114, 42806: 17, 68420:
54, 6284: 90, 40268: 53, 6482: 34, 62804: 115, 60248: 105, 40286: 84,
82064: 93, 42860: 64, 20846: 106, 26480: 4, 80246: 10, 62840: 47, 40826:
29, 60284: 20, 40682: 118, 20864: 67, 86402: 37, 80264: 60, 28046: 88,
86420: 101, 60824: 119, 48026: 94, 40862: 112, 28064: 74, 2468: 42,
60842: 19, 8624: 45, 2486: 103, 68024: 5, 28604: 66, 48062: 33, 26048:
25, 8642: 23, 62408: 59, 68042: 11, 46028: 16, 48206: 117, 48602: 50,
28640: 46, 26084: 38, 82406: 1, 48620: 87, 6824: 73, 20468: 111}))
('numbers:', 120)
```