Tutorial on Python Programming

Indian Institute of Astrophysics, Bangalore, India

Chetan Giridhar
(http://technobeans.com)

What to expect

- This tutorial will introduce core Python language to you in a decent detail.
- After going through the posts and practising stuff, you will be comfortable enough to start writing programs for basic tasks.
- This is a Learner to Learner series. Please ask queries, share more about topic of discussion and point to the mistakes, so that we all learn together.
- Facilitation & guidance.

Python - What, Why and How?

- Python is a powerful scripting language created by Guido Van Rossum in 1998.
- Python in itself is a feature rich, object oriented language and can be used for Rapid Application Development of medium sized applications.
- Python is a scripting language of choice of a large number of the security professionals, developers and automation engineers. There is an ever-growing community supporting Python.
- You can get Python as a free download from Official Python website or as ActivePython from ActiveState website. There are other versions available, but the mentioned ones solve the purpose.

Python Features...

- It is free and open source.
- Easy coding - It's meant for Rapid application development. Consistent style of coding and usage make your life easier than remembering some 100 shortcuts or tricks.
- Excellent Readability and Maintainability - Proper Indentation of code is not a choice, rather the way of Python coding. If you pick Python code by different programmers, you can be sure to see similar looking code.
- It is Object-oriented - OOP is not a patch work for Python, rather an in-built feature.
- Python include the use of transparent byte-code compilation for speed, automatic memory management and garbage collection, and a very powerful object oriented and modular design.

- Python ships with a large number of modules.
- No semi-intelligence - No ambiguous type-conversions or assumptions.
- Exception Handling is built into the language.
- It is Cross Platform compatible.

Hello World!

- Programs in Python can be run in two ways:
  - Executing a Python script
  - Executing Python code interactively

- Helloworld.py

```
# Script starts
print "Welcome to Technobeans!"
# Script ends
```

- Now save this script in “C:\” directory, and run the script from DOS prompt as: python hello.py

Command Line Arguments

- Sys module comes to the rescue.
- Import sys
- Print sys.argv[0]
- For i in sys.argv[]:

  print i
- The type of all the command line arguments is **str.**
- Checking for input data types and type conversion should always be done.

Python Interpreter

- Executing interactively: Open **DOS prompt** and just type **python**.
- C:\>python ActivePython 2.5.1.1 (ActiveState Software Inc.) based on Python 2.5.1 (r251:54863, May 1 2007, 17:47:05) [MSC v.1310 32 bit (Intel)] on win32 Type "help", "copyright", "credits" or "license" for more information. >>>
- >>> print "Welcome to Technobeans!" Welcome to Technobeans!
- To come out of the interactive shell, press **Ctrl+Z** and **Enter** key.
- When Python interpreter loads, modules/packages will be available for importing <Python installation>\lib\site-packages.
- sys.path.append("C:****My_Scripts") – for importing user defined modules.

.pyc and .pyo

- Python source code is automatically compiled into Python byte code by the CPython interpreter. Compiled code is usually stored in PYC (or PYO) files, and is regenerated when the source is updated, or when otherwise necessary.
- Automatic compilation – importing a module. But the module gets executed.
- Explicit compilation – py_compile.compile("module.py") – generates module.pyc
- When the Python interpreter is invoked with the -O flag, optimized code is generated and stored in '.pyo' files.
- Passing two -O flags to the Python interpreter (-OO) will cause the bytecode compiler to perform optimizations that could in some rare cases result in malfunctioning programs.

- A program doesn't run any faster when it is read from a ‘.pyc’ or ‘.pyo’ file than when it is read from a ‘.py’ file; the only thing that's faster about ‘.pyc’ or‘.pyo’ files is the speed with which they are loaded.
- When a script is run by giving its name on the command line, the bytecode for the script is never written to a ‘.pyc’ or ‘.pyo’ file. Thus, the startup time of a script may be reduced by moving most of its code to a module and having a small bootstrap script that imports that module. It is also possible to name a‘.pyc’ or ‘.pyo’ file directly on the command line.
- The module ‘compileall’{} can create ‘.pyc’ files (or ‘.pyo’ files when -O is used) for all modules in a directory.
- To distribute a program to people who already have Python installed, you can ship either the PY files or the PYC files.
- Convert - .py to .exe -> Py2exe helps.

Data types available in Python

- The important (or rather the most commonly used) base data types in Python are **Numeric Types** (numbers - **int**, **long**, **float**), **Sequence Types** (**string**, **list**, **tuple**),**Mapping Types**(**dictionary**) and **Boolean(True/False)**.

- For user-defined types, one has to declare a class.

- Out of the mentioned types, we will discuss all the **Numeric Types**, **string** (Sequence Type) and **Boolean**. Others will be diccussed once we have learnt control structures and some built-in functions. The **string** data type will be revisited as well.

Declare a variable of a particular type?

- '=' operator is the key.
- You do not have to use any special defining methods. You simply say: var_name = literal, where **var_name** is the name you choose for **variable** and **literal** is a constant value of any data type.
- What's presnet to the right defines the type of LHS variable name.
- Everything in Python is an object. Python finds the type of a variable from the value to which points.
- Also, because there is no strict defining style, you can point var_name to any other literal of a different data type.

How do I find the data type

- 2 ways of doing it:
  - Type()
  - Isinstance(variable, type)

```
Var = 34
Print type(var)

if isinstance(var, int):
    print True
```

Getting User Input

- raw_input() function is used.
- Python accepts whatever the end user types till he or she presses the “Enter” key, as an input and assigns it to the variable on left side of the assignment.
- >>> a_var = **raw_input**("Please enter your name:")

  Please enter your name:Tester

  >>> a_var 'Tester' >>>
- Whatever comes from STDIN is always a stream of **characters**. In your program, you have to convert the string to the type you want and in case of conversion error, inform the user about the incorrectness.
- None as NO Input?

None, Empty!!

- Def foo():
  pass
  print foo() - None
- list = []
  for i in list:
  print i   - Empty
- a = raw_input(“enter a:”)
  press “enter key” - Empty/ Sometimes referred as Nothing
- None is commonly used for exception handling.

Documentation Strings

- The first line should always be a short, concise summary of the object's purpose.
- If there are more lines in the documentation string, the second line should be blank, visually separating the summary from the rest of the description.
- def example():

  """This is just an example.

  It does something. """

  print "In example"
- print example.__doc__

Operators

- Addition: +
- Subtraction: -
- Multiplication: *
- Exponentiation: * *
- Division: / and / / (floor or [x])
- Modulus: %

Operators

- Greater than: >
- Less than: <
- Greater than or equal to: >=
- Less than or equal to: ⇐
- Equal to: ==
- Not equal to: <> !=

Conditional Execution
if / if-else / if-elif-if

- if condition1:
  if condition2:
  True path
  else:
  False path
  else:
  False path

Looping Execution - while / for

- while condition :

  Statements
- for var_name in Sequence/function which outputs a sequence:

  statements

  Range() - iterate over a sequence of numbers.

  range(5) = [0,1,2,3,4]

  range(5,10) = [5,6,7,8,9]

  range(0,10,3) = [0,3,6,9]
- Do-while: Not available.

  While True:

  if condition:

  break

Break, Continue and Pass

- **Break:** The loop in which this statement is found exits as soon as it is reached. The control then jumps to the outer loop in case of nested loops or to the main script.

- **Continue:** The loop in which this statement is found skips rest of the statements found after this, in its body. The control then jumps to the beginning of the same loop and the next iteration is started.

- **Pass:** The pass statement does nothing. It can be used as a place-holder for a function or conditional body when you are working on new code.

Sequence data type

- A **Sequence type** in python is a mini built-in data structure that contains elements in an orderly manner which can be fetched using indices. There are three types of sequences:

  Strings >>> a_string = “Chetan Giridhar"

  Lists >>> a_list = [“Chetan",”Giridhar"]

  Tuples >>> a_tuple = (“Chetan",”Giridhar")

- **Strings** and **tuples** are immutable in Python, but **lists** are mutable. (An immutable object can not modified-in-place. What it means is that when a function/expression tries to modify its contents right there its original memory location, it either fails or creates an altogether new object.).

  >>> a_string [0] = "t“ - Invalid

  >>> a_list[0] = "My Example“ - Valid

  >>> a_tuple[0] = "My Example“ - Invalid

Operator Overloading

Python supports operator overloading.

- Indexing - Getting an element with the help of an integer index.
- Slicing - Getting a sub-sequence by using a special syntax of lower and upper index.
- Appending/Extending/Concatenating - Adding elements at the end of a sequence.
- Deleting an element - For first occurrence.
- Modifying an element - At a given index.
- Sorting - Based on element types.

Working with lists

- Defining lists

  numList = [2003,2005,2008,2010]

  strList = [“IIA”, “Chetan”, “Python”]
- Accessing a list

  For x in numList:

  ```
      print x
  ```

  print strList[0] or strList[-1]
- Slicing a list

  firstHalf = numList[:2]

  lastHalf = numList[2:3]
- Adding and removing items

  list.append(“2009”) – end of the list

  list.extend(list1) – end of the list

  list.insert(index, item) – at a specified index

Pop(index) – removes the element at the index

remove(item) – removes the first occurrence of item.

- Sorting a list

  list.sort()

  list.reverse()

  list = ["iia","IIA", "chetan", "python"]

  list.sort(key = str.lower)

  for i in list:

  print I

- Converting tuple to list

  List(tuple)

String functions

- test = ‘This is a simple string’
- len(test) = 23
- test.count(‘r’) = 1
- test.find(‘r’) = 18
- test = test.replace(‘simple’, ‘short’)
  ‘This is short string’
- test.index(‘simple’) = 10
- test.split(‘is’) = [‘This ‘, ‘a short string’]
- ‘some’. join(test.split(‘is’))
- test.upper() and test.lower() and test.lower.capatalize()
- test.lstrip(‘ ‘) and test.rstrip(‘\t’)

eval vs exec

- Exec function will execute Python Code that is contained in str string and return the result.
- Eval works similar to the exec function except that it only evaluates the string as Python expression and returns the result.
- def foo():

  print "foo“

  eval("foo" + "()")

- cards = ['king', 'queen', 'jack']

  codeStr = "for i in cards:\

  print i"

  exec(codeStr)

What is a Python dictionary?

- A Python **dictionary** is a **Mapping Type**, which is a mini data-structure, containing key-value pairs.

- A **dictionary** has unsorted key-value pairs. It means the data in a **dictionary** is not in any necessary order, unlike a sequence type. This helps to give Python, a lot of freedom in terms of memory management.

- The **key** in a **dictionary** should be immutable.

- A **key** is always unique. If you use the same **key** in an assignment, the value of this **key** gets updated, instead of adding a new **key** with the same name.

Working with dicts…

```
>>> aDict = {"a" : 1, "b": 3}
>>> type(aDict)
    type 'dict‘
>>> aDict.keys()  and  aDict.values()
>>> aDict.items()    -   [('a', 1), ('b', 2)]
>>> for key in aDict.keys():
            print "Key: " + str(key) + " Value: " +str(aDict[key])
    Key: a Value: 1 Key: b Value: 2
>>> aList = [1,2,3]
    aDict = {"a" : 1, "b": 3}
    aNestedDict = {'key1' : aList, 'key2': aDict}
>>> aNestedDict['key2']['a']    -    1
>>> for key,value in aDict.iteritems()
    swapDict[value] = key
```

Code Reusability in Python

- Implementing code reusability happens through the use of functions, modules and packages.
- You write the code in generic ways to come up with functions, pack related functions in a single module (file) and pack multiple related modules inside a package (directory structure).
- Another form of the above is that you write methods (functions) that belong to a class (that signifies a user defined data type), put related classes inside a module, put related modules inside a package.
- Everything is an object in Python. It would make sense that you create an object (of a class) to talk to another object (built on or your own) rather than coding in the traditional way while being in the world of a wonderful OO language like Python.

Working with functions

- A typical function

  def multiply(operand1,operand2):

  return a * b

  The above is a function with name multiply. What it does is, it takes two arguments and returns the product of them.

  Here operand1 and operand2 are formal parameters. Note that we did not define their type in the function definition. So it becomes the responsibility of the function body to do type checking.

  The *execution* of a function introduces a new symbol table used for the local variables of the function. More precisely, all variable assignments in a function store the value in the local symbol table; whereas variable references first look in the local symbol table, then in the global symbol table, and then in the table of built-in names. **LGB** is the mantra.

  Default values can be used and passed.

Scoping of variables

- In Python, variables that are only referenced inside a function are implicitly global. If a variable is assigned a new value anywhere within the function's body, it's assumed to be a local. If a variable is ever assigned a new value inside the function, the variable is implicitly local, and you need to explicitly declare it as 'global'.
- Though a bit surprising at first, a moment's consideration explains this. On one hand, requiring global for assigned variables provides a bar against unintended side-effects. On the other hand, if global was required for all global references, you'd be using global all the time. You'd have to declare as global every reference to a built-in function or to a component of an imported module. This clutter would defeat the usefulness of the global declaration for identifying side-effects.

Rules – making a function call

1. Number of arguments - Number of arguments should be equal to the number of formal parameters used in the definition of the function.
2. Type of arguments - The argument and the corresponding parameter should have the same data type.
3. Order of arguments - The arguments should have the same order as that of the formal parameters, otherwise you might get type mismatch errors or unexpected results in some cases.

Example:

```
>>>def printData(name, age):
        return "Name: %s Age: %d" % (name,age)
>>> printData("Tester",100) 'Name: Tester Age: 100
```

if you carefully see the string formatting used, %s and %d format specifiers make Python take care of type checking automatically.

Pass By – Value and reference

- Parameters are passed by value.
- Strings and Tuples are immutable and hence cant be changed.
- When lists are passed, they get muted and hence it looks as if they are getting passed by references.

Lambda – Anonymous functions

- Anonymous functions: functions that are not bound to a name at run time.
- g = lambda x: x**2

  g(8) – 64
- Lambda definition does not include a "return" statement -- it always contains an expression which is returned.
- def make_incrementor (n): return lambda x: x + n

  The above code defines a function "make_inrementor" that creates an anonymous function on the fly and returns it. The returned function increments its argument by the value that was specified when it was created.
- f = make_incrementor(2) . f(42) = 44.

- >>> foo = [2, 18, 9, 22, 17, 24, 8, 12, 27]

- >>> print **filter**(lambda x: x % 3 == 0, foo)
- [18, 9, 24, 12, 27]

- >>> print **map**(lambda x: x * 2 + 10, foo)
- [14, 46, 28, 54, 44, 58, 26, 34, 64]

- >>> print **reduce**(lambda x, y: x + y, foo)
- 139

File Handling

- **open(filename, mode)**

  >>> f = open('/tmp/workfile', 'w')

  >>> print f

  <open file '/tmp/workfile', mode 'w' at 80a0960>

  Modes: r, w, a, b, r+, rb, wb, r+b
- **f.read(size)**

  When *size* is omitted or negative, the entire contents of the file will be read and returned;

  If the end of the file has been reached, f.read() will return an empty string ("").
- **f.readline()** reads a single line from the file; a newline character (\n) is left at the end of the string, and is only omitted on the last line of the file if the file doesn’t end in a newline.

- **f.readlines()** returns a list containing all the lines of data in the file. If given an optional parameter *sizehint*, it reads that many bytes from the file and enough more to complete a line, and returns the lines from that.
- **Read words:** Memory efficient, fast, and simpler code:

```
for line in f:
        print line OR for word in line.split()
```

- **f.write(string)** writes the contents of *string* to the file, returning None. To write something other than a string, it needs to be converted to a string first.
- **f.tell()** returns an integer giving the file object's current position in the file, measured in bytes from the beginning of the file.
- To change the file object's position, use **f.seek(offset, from_what).**
- **f.close()** to close it and free up any system resources taken up by the open file.
- **F.endswitch("py")** search for files with extension py – True/False

Some more file operations

- Import os
- os.remove(filepath)
- os.rename(oldzfile, newFile)
- os.listdir(dirPath)
- os.walk(dirPath)

Pickling

- Way to achieve object serialization in Python.
- Pickling is used when you want to save more complex data types like lists, dictionaries, or class instances in a file.
- Dumping and parsing logic is difficult to develop and maintain.
- Python **cPickle** module can take almost any Python object and convert it to a string representation; this process is called *pickling*.
- pickle is the standard way to make Python objects which can be stored and reused by other programs or by a future invocation of the same program
- Reconstructing the object from the string representation is called *unpickling*.
- Between pickling and unpickling, the string representing the object may have been stored in a file or data, or sent over a network connection to some distant machine.
- If you have an object x, and a file object f that's been opened for writing, the simplest way to pickle and unpickle the object is: pickle.dump(x, f) and x = pickle.load(f).

Some Concepts

- code block: A code block is a piece of Python program text that can be executed as a unit, such as a module, a class definition or a function body.
- Execution frame: Every code block is assigned an execution frame by Python.

  1. Python maintains some debugging information about the code block as a part of the execution frame.

  2. The execution frame determines the control flow after the execution of the code block completes.

  3. It defines namespaces for the code block
- Namespace: A namespace is like a dictionary. You can consider the names (identifiers/variable names) as keys of this dictionary and the corresponding values to be the objects to which they point to.

Regular Expressions

- Regular expressions are a very powerful tool in any language. They allow patterns to be matched against strings.
- Regular expressions in Python are handled by using module ‘re’.
- Import re

  test = 'This is for testing regular expressions in Python.‘
- result = re.**search**('(Th)(is)',test)

  print result.group(0), result.group(1)
- result = re.**match**(‘regular’, test)

  print result – None (match only at the beginning of string)
- ourPattern = re.compile ('(.*?)(the)')
  testString = 'This is the dog and the cat.‘
  result = ourPattern.match (testString)
  result.group (0)

- Regular expressions use the backslash character ('\') to indicate special forms or to allow special characters to be used without invoking their special meaning.
- What if we want to use ‘\’ in file path?
- The solution is to use Python’s raw string notation for regular expression patterns; backslashes are not handled in any special way in a string literal prefixed with 'r'.
- f = open(r"c:\Windows\notepad.exe", "r")
- Special characters: ‘.’, ‘^’, ‘$’,’*’,’+’,’?’,{m}, {m,n},’\’, ‘|’, ‘(…)’, ‘\d’, ‘\D’, ‘\s’, ‘\S’, ‘\w’, ‘\W’

- re.**split**('[a-f]+', '0a3B9’) - ['0', '3', '9']

- someString = 'I have a dream.'
  print re.**sub** ('dream', 'dog', someString)
  print someString
- Regular Expressions don’t change the actual string.
- Regular Expressions are very helpful but resource intensive. It should not be used for matching or finding simple texts.

Some Concepts…

- An execution frame creates two kinds of namespaces for a code block, namely local and global. The third type of namespace is called the built-in namespace and is created when the Python interpreter loads.
- binding and unbinding:

    1. When you define a function, you bind the formal parameters.
    2. You bind a name when you use an import construct (to the namespace of the block which contains the import statement).
    3. When you define a class or a function, using the class and def keywords, you bind the names of the functions and classes and they are available in the containing block (e.g. a module)
    4. When you do an assignment operation
    5. The loop control variable in a for loop construct creates a dynamic binding
    6. In an except clause, when you provide argument to the clause

Modules

- How do I use an existing module and the functions and classes contained in it?

  To use an existing module, you need to first create a reference to it in the current namespace. In simple words, you would need to import a module to use its functionality.
- How do I know whether a module is available in current namespace?

  You can use **dir()** built in command to check what all reference variables are available to you in current namespace. You can go one step ahead and use **dir()** to see what references are available in a particular object's (e.g. module's) namespace by passing object as the argument.
- import random

  dir(random)

Modules…

- Importing Modules:

  import random

  from random import randint

  from random import *

  import random as Rand
- Access classes and functions of module:

  For this module name and the **dot** operator should be used.
- Importing modules in blocks?
- From <modulename> import <functionanme>

OOP in Python

- Python as a language is built in object oriented way. Everything is an object in Python and it really means so.
- When compared with other Object Oriented languages like C++ and Java, you will observe that most of the features are available in Python, in a much simpler way. You will also observe that many of the access related features (and those loads of keywords and rules) are not there.
- Inheritance
- Multiple inheritance
- Abstraction
- Polymorphism
- Overriding
- Operator Overloading
- Object Instances

Classes in Python

- Creating a class:

  class MyClass:

  pass

- Instantiating a class:

  a = MyClass()

  print a

  <__main__.MyClass instance at 0x01206A08>

  Yes, it does look like a function call, in fact you can very well have a function and a class with the same name in your Python code. The syntax for instantiation and function call will look exactly the same.

  Python understands the type of any variable based on the object to which it points. So, if **MyCode** points to the body of a class, it **a = MyCode()** instantiates a class and if it points to a function body, the function is called and a holds its return values.

A more meaningful Class

- class SimpleClass:

```
    def __init__(myVal):
        self.myVal = myVal
    def getValue(self):
        return self.myVal
    def setValue(self, newVal):
        self.myVal = new Val
```

- The class **SimpleClass** has three methods inside it.
- The first one has the name **_ _init_ _** and is called the constructor. The name of this method should always be **_init_ _**. It takes the argument **myVal**. This value is used to initialize an object. This method is used frequently to initialize an object with a given state, rather than calling a whole set of functions to set its state after creation.

- **self** is a convention used by Python programmers to depict the current object reference. Unlike many other OOP languages, the current object reference has to be explicitly defined in Python class definitions - instance variables and methods. The first argument of every method has to be self. This argument is used only while defining the method, in the method call, it is omitted.
- Inside the **_ _init_ _**, it says **self.myVal = myVal**. This is a simple assignment operation. Here **self.myVal** defines an instance variable (object variable).
- We see two other methods, which look like normal functions. The difference being that they are bound to an object and there is essentially a lot of information which need not be passed to them as arguments every time, as they have access to the object variables and class variables (we'll discuss about class variables later).

- **getValue()** returns the value of the **self.myVal** and **setValue()** sets it to the value that is passed.
- **get** and **set** functions are called accessors and mutators respectively. They have a lot of utility in preventing the private variables of a class from getting modified inaccurately or unexpectedly.
- simpleClassObject = SimpleClass(12) print "Value: " + **str**(simpleClassObject.getValue()) simpleClassObject.setValue(15) print "Value: " + **str**(simpleClassObject.getValue())
- Value: 12

  Value: 15

```
class A:
        def __init__(self):
                print "Called A"
        def __local(self):
                print "In local"
        def load(self):
                print "Loaded from A"
                self.__local()
class B():
        def __init__(self):
                print "Called B"
        def load(self):
                print "Loaded from B“
class C(B,A):
        def __init__(self):
                a = A()
                        print "In C“
c = C()
c.load()
if hasattr(C, "load"):
        print "yes"
```

Static analysis of code

- PyChecker is a static analysis tool that finds bugs in Python source code and warns about code complexity and style.
- Pylint is another tool that checks if a module satisfies a coding standard, and also makes it possible to write plug-ins to add a custom feature.
- Pylint more popularly used.

Exception Handling

- The statements used to deal with exceptions are **raise** and **except**.
- **def** throws():

  raise RuntimeError('this is the error message')

  def main():

  throws()

  if __name__ == '__main__':

  main()
- Traceback (most recent call last): File "throwing.py", line 10, in <module> main()

  File "throwing.py", line 7, in main throws()

  File "throwing.py", line 4, in throws raise RuntimeError('this is the error message')

  RuntimeError: this is the error message

- f = None
  try:
    f = open("aFileName")
    f.write(could_make_error())
  except IOError:
    print "Unable to open file"
  except: # catch all exceptions
    print "Unexpected error"
  else: # executed if no exceptions are raised
    print "File write completed successfully"
  finally: # clean-up actions, always executed
    if f:
     f.close()
- Raise – re-raises the same exception thrown and dies.

Garbage Collection

- Python's memory allocation and de-allocation method is automatic.
- Python uses two strategies for memory allocation **reference counting** and **garbage collection**.
- Reference counting works by counting the number of times an object is referenced by other objects in the system. When references to an object are removed, the reference count for an object is decremented. When the reference count becomes zero the object is de-allocated.
- Caveat is that it cannot handle *reference cycles.*
- **def** make_cycle():

  l = [ ]

  l.append(l)

- Garbage collection is a scheduled activity. Python schedules garbage collection based upon a threshold of object allocations and object de-allocations.
- import gc

  print gc.get_threshold()

  (700,10,10) - This means when the number of allocations vs. the number of de-allocations is greater than 700 the automatic garbage collector will run.
- The garbage collection can be invoked manually in the following way:

  import gc

  gc.collect()
- gc.collect() returns the number of objects it has collected and deallocated.

Logging

- Logging is performed by calling methods on instances of the Logger class.
- The methods are debug(), info(), warning(), error() and critical(), which mirror the default levels.
- **import logging**

  LOG_FILENAME = 'example.log'
  logging.basicConfig(filename=LOG_FILENAME,level=logging.DEBUG)

  logging.debug('This message should go to the log file')
- File Contents:

  DEBUG:root:This message should go to the log file.
- Messages with higher precedence get logged to the LOG_FILENAME.

Python in Astronomy

- pFitsio or PyFits and the NumPy, SciPy, astronomical images and tables can be easily accessed and manipulated as numerical arrays.
- BoA (Bolometer Analysis Package)
- Astro-WISE (widefield imaging system)
- astLib – Python Astronomy Modules - astronomical plots, some statistics, common calculations, coordinate conversions, and manipulating FITS images with World Coordinate System (WCS) information through PyWCSTools

References

- Python.org
- Doughellmann.com
- Devshed.com
- Testingperspective.com
- Network-theory.co.uk
- Boodebr.org
- Stackoverflow.com

Thanks ☺