

Gentle Introduction to Python

Tom Kuiper

November 2, 2014

1 Introduction

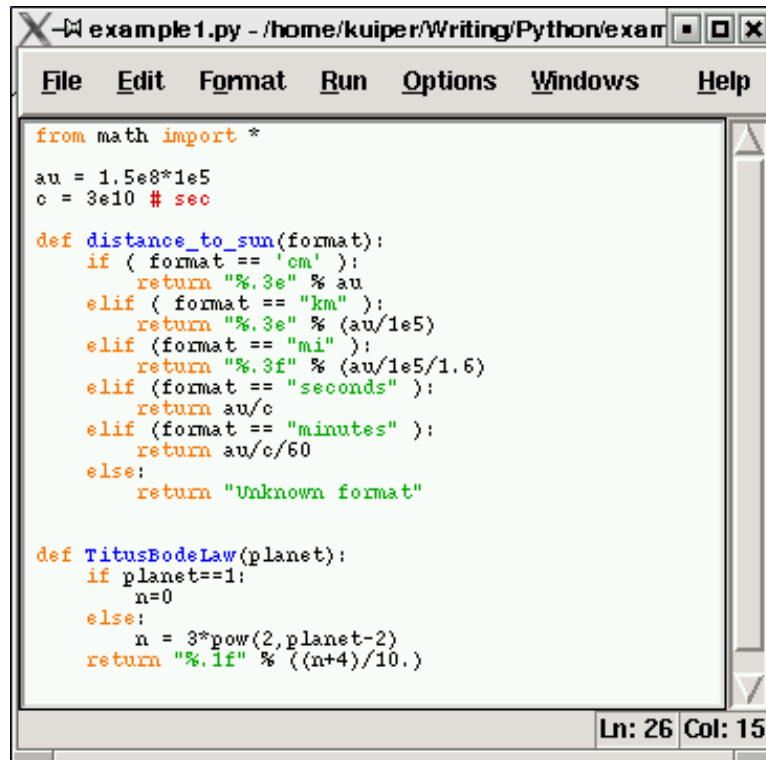
Python is a very elegant, syntactically simple scripting language. However, it has, in some circles, a reputation for being obscure. Indeed, code written by a computer science graduate may be almost unreadable by someone whose background is in procedural languages like C, FORTRAN, IDL, Visual Basic, etc. Nevertheless, Python can be used as a procedural language. In time, one can grow into the more sophisticated features.

2 Python Basics

2.1 A Simple Introduction

The functions given in Figure 1 should be easily understood by anyone with some programming experience. The points to note in `distance_to_sun()` are

1. Python only uses functions. (If Python functions are created from C or FORTRAN, then subroutines are converted to functions, that is, the modified arguments are returned as output.)
2. The type of a variable is implicit in the value it is assigned. The variables `au` and `c` are type `float`. `format` is type `string`. There are ways to create variables of one type from another type. For example `"%.3e" % (au/1e5)` converts a `float` to a `string`.
3. The start of a block of code is indicated by a colon and any decent editor, knowing that it is working on a `.py` file, will automatically indent the next line.
4. A block of code ends with a return to the previous level of indentation. Some keywords, like `return`, by definition end a block and the editor will revert to the previous level.
5. Output formatting is a little odd but very similar to C except that `%` separates the format string from the list of variables to be formatted, separated by commas and enclosed with parentheses.



```
from math import *

au = 1.5e8*1e5
c = 3e10 # sec

def distance_to_sun(format):
    if (format == 'cm'):
        return "%.3e" % au
    elif (format == "km"):
        return "%.3e" % (au/1e5)
    elif (format == "mi"):
        return "%.3f" % (au/1e5/1.6)
    elif (format == "seconds"):
        return au/c
    elif (format == "minutes"):
        return au/c/60
    else:
        return "Unknown format"

def TitusBodeLaw(planet):
    if planet==1:
        n=0
    else:
        n = 3*pow(2,planet-2)
    return "%.1f" % ((n+4)/10.)
```

Ln: 26 Col: 15

Figure 1: Two sample Python functions. (The height of the window was adjusted to remove blank space.)

2.2 Trying it out with Idle

idle is a programming environment bundled with Python. Start idle, open a new window from the File menu. You will then have the window shown in Figure 1. Type in the code for `distance_to_sun()`, save it and run it. Run is on the menu bar. In this case, Run produces no output. It only defines the variables `au` and `c` and the functions `distance_to_sun()` and `TitusBodeLaw()`.

In the idle main window (Figure 2), verify the function. Note that the some of the outputs are strings while others are type `float`. The strings can be converted back into type `float` by

```
>>> float('1.500e+08')
150000000.
```

The above also illustrates that the Idle main window provides a Python command input. The Python command line can useful by itself for simple calculations.

```
myself@somehost:~$ python
```

```

Python 2.3.5 (#2, Oct 16 2006, 19:19:48)
[GCC 3.3.5 (Debian 1:3.3.5-13)] on linux2
Type "copyright", "credits" or "license()" for more information.

*****
Personal firewall software may warn about the connection IDLE
makes to its subprocess using this computer's internal loopback
interface. This connection is not visible on any external
interface and no data is sent to or received from the Internet.
*****

IDLE 1.0.5
>>> ===== RESTART =====
>>>
>>> distance_to_sun('cm')
'1.500e+13'
>>> distance_to_sun('km')
'1.500e+08'
>>> distance_to_sun('mi')
'93750000.000'
>>> distance_to_sun('seconds')
500.0
>>> distance_to_sun('minutes')
8.333333333333333

```

Figure 2: Test of the function shown in Figure 1.

```

Python 2.3.5 (#2, Oct 16 2006, 19:19:48)
[GCC 3.3.5 (Debian 1:3.3.5-13)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> year = 365.25*24*60*60
>>> print "%.5e sec" % year
3.15576e+07 sec
print "%.5e sec" % (year/pi)
1.00451e+07 sec

```

2.3 Lists

Lists are a generalized version of C and FORTRAN arrays. The following behaves like a C array (indexed from 0 to 9).

```

>>> distance = []
>>> for i in range(1,10):
distance.append(TitusBodeLaw(i))

>>> print distance
['0.4', '0.7', '1.0', '1.6', '2.8', '5.2', '10.0', '19.6', '38.8']
>>> print distance[2]
1.0

```

Things to note:

1. When using the command prompt, a colon automatically starts a block which ends when two linefeeds are entered. The block is then executed.
2. `range(start,stop,interval)` function takes the place of the `for start to stop by interval` construct in most languages. The argument `stop` is required. The default for `start` is 0. If a third argument is given, it is `step`.
3. In fact, `range()` produces a list.

```
>>> range(2,10,2)
[2, 4, 6, 8]
```

Note that 10 is not in the list, which would not be consistent with indexing from 0.

4. One might expect that adding an item to the end of a list could be done with a function `append(mylist,item)`. However, `append` as a keyword might very well appear elsewhere in a large program with a different meaning. The dot construct indicates that this `append` function belongs to the list data type only¹.

Lists can be very general, in that each item can be any kind of variable, even another list. For example,

```
>>> Mercury = [0.39,[2439,2439],0.05527,58.6462,'']
>>> Venus = [0.72,[6051,6051],0.81499,243.01,'Cytherian']
>>> Earth = [1.00,[6378.140,6356,775],1.00,0.99726966,'Terrestrial']
```

Of course, this is not the clearest way of encoding this information, since the program has to know the meaning of the variable in each place in the list.

2.4 Dictionaries

Dictionaries are a much better way to encode diverse information. Consider this:

```
>>> Mercury = {'au':0.39,'radii':[2439,2439],'mass':0.05527,
... 'sid.rot.pd':58.6462,'genetive':''}
>>> Mercury.keys()
['mass', 'radii', 'au', 'genetive', 'sid.rot.pd']
```

1. Python will give a continuation prompt of `"..."` when an input is incomplete.
2. The data type (*class*) `dictionary` has a function (*method*) `keys` which can be quite useful to see what data are defined in the dictionary.

¹For those who intend to swim in deeper waters, let's note here that the *object* `mylist` is an *instance* of the *class* `list`. `mylist.append()` is a *method* of the class `list`

2.5 Tuples

A tuple is another *sequence* data type.

```
>>> ra = (5+(32+47.0/60)/60)*pi/12
>>> dec = -(5+(24+28.0/60)/60)*pi/180
>>> x = cos(ra)*cos(dec)
>>> y = sin(ra)*cos(dec)
>>> z = sin(dec)
>>> vector = x,y,z
>>> print vector
(0.11794886238916301, 0.98853742292222613, -0.094243457828042942)
```

You can't change the items of a tuple (well, you can in a complicated way) but unpacks more cleanly than a list.

```
>>> vector[1] = 5
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: object doesn't support item assignment
>>> d1,d2,d3 = vector
>>> print d2
0.988537422922
```

This suggests that in some cases it may be convenient to have a function return a tuple:

```
>>> def direction_cosines(ra,dec):
...     x = cos(ra)*cos(dec)
...     y = sin(ra)*cos(dec)
...     z = sin(dec)
...     return x,y,z
...
>>> d1,d2,d3 = direction_cosines((5+(32+47.0/60)/60)*pi/12,\
                                -(5+(24+28.0/60)/60)*pi/180)
(0.11794886238916301, 0.98853742292222613, -0.094243457828042942)
```

3 Modules

Modules group functions (and classes) by topic, such as numerical methods, graphics, scientific disciplines, etc. There are different ways to access functions that are in modules.

3.1 Importing Code from Modules

3.1.1 from *module* import *

In the example in Figure 1 we passed over a key statement:

```
from math import *
```

`math` is a module that provides the basic mathematical functions. In this example, all the data and functions in `math` were brought into current context, which may be that of the main command line or another module. In fact, the code shown in Figure 1 belongs implicitly to a module called `example1` because it was saved in a file `example1.py`.

3.1.2 The Search Path

The `import` command searches all the directories included in `sys.path` starting with the current directory.

```
>>> import sys
>>> print sys.path
['', '/usr/lib/python23.zip', '/usr/lib/python2.3', \
'/usr/lib/python2.3/plat-linux2', '/usr/lib/python2.3/lib-tk', \
'/usr/lib/python2.3/lib-dynload', \
'/usr/local/lib/python2.3/site-packages',
'/usr/lib/python2.3/site-packages', \
'/usr/lib/python2.3/site-packages/HTMLgen', \
'/usr/lib/python2.3/site-packages/Numeric', \
'/usr/lib/python2.3/site-packages/PIL', \
'/usr/lib/python2.3/site-packages/gtk-2.0']
```

The path can be expanded by setting the environment variable `PYTHONPATH` using the colon-separated syntax of `PATH`. These directories are inserted between the current directory and the standard search path.

```
kuiper@nutmeg:~$ export PYTHONPATH=/home/kuiper:/home/kuiper:tmp
kuiper@nutmeg:~$ python
Python 2.3.5 (#2, Oct 16 2006, 19:19:48)
[GCC 3.3.5 (Debian 1:3.3.5-13)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import sys
>>> print sys.path
['', '/home/kuiper', '/home/kuiper/tmp', '/usr/lib/python23.zip', \
'/usr/lib/python2.3', '/usr/lib/python2.3/plat-linux2',
....
]
```

3.1.3 import module

Another way to import data and functions from module would be as shown here.

```
kuiper@nutmeg:~$ python
Python 2.3.5 (#2, Oct 16 2006, 19:19:48)
[GCC 3.3.5 (Debian 1:3.3.5-13)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
```

```
>>> import math
>>> dir(math)
['__doc__', '__file__', '__name__', 'acos', 'asin', 'atan', 'atan2', \
'ceil', 'cos', 'cosh', 'degrees', 'e', 'exp', 'fabs', 'floor', \
'fmod', 'frexp', 'hypot', 'ldexp', 'log', 'log10', 'modf', 'pi', \
'pow', 'radians', 'sin', 'sinh', 'sqrt', 'tan', 'tanh']
```

(Note the useful `dir()` function.) In this case, the module becomes accessible but not as an integral part of the current environment. This would be useful if one needed to have other definitions for `e` and `pi`.

```
>>> e = 1.6e-19 \# coulomb
>>> pi = 8.8 \# parallax
>>> print math.pi
3.14159265359
>>> print math.e
2.71828182846
```

3.1.4 import *package as alias*

If a package has a long name, it could be tedious to use the whole name every time. In that case, the following may be more convenient:

```
>>> import datetime as D
>>> dir(D)
['MAXYEAR', 'MINYEAR', '__doc__', '__file__', '__name__', 'date', \
'datetime', 'time', 'timedelta', 'tzinfo']
>>> print D.date.today()
2007-06-02
```

This aliases the module `datetime` to `D`.

3.2 Foreign Modules and OO Programming

If you write all your own code, Python does not prevent you keeping to your procedural programming habits. However, there are very many useful Python modules which are either built-in or are contributed by other programmers. Almost invariably they are written in the object-oriented paradigm. The built-in `datetime` module is a good example. The last command in the above example means, “Print the result of invoking *method* `today` of the (object) *class* `date` in the module `D`.”

To the procedural programmer, the information available from the help function is so obscure as to be almost useless. However, we now know from the above example that there are object classes called `date`, `datetime`, etc. in the `datetime` module. A class is a prototype object. In FORTRAN we might think of `double` as a class and `pi` as an object which is an instance of that class.

We can scan the help on the class `date` to discover that it has a method `today`. Below we create an object `d1` of the class `date` in module `D` (= `datetime`) and invoke its method `today()`.

```
>>> d1 = D.date
>>> d1.today()
datetime.date(2007, 6, 2)
>>> print d1.today()
2007-06-02
```

Now we create an object obtained from the method `today` and invoke some of its methods.

```
td = d1.today()
>>> print td
2007-06-02
>>> td.weekday()
5
>>> td.isoweekday()
6
>>> td.isocalendar()
(2007, 22, 6)
>>> td.isoformat()
'2007-06-02'
>>> td.ctime()
'Sat Jun  2 00:00:00 2007'
```

`isocalendar()` returns the year, week number, and day of the week from object `td`. In the case of method `ctime`, the time is 00:00:00 because `td` is a `date` object instead of a `datetime` object.

We can obtain *attributes* of the *object* `td`:

```
>>> td.year
2007
>>> td.month
6
>>> td.day
2
```

With these clues, the goobledygook returned from

```
>>> help(D.date)
```

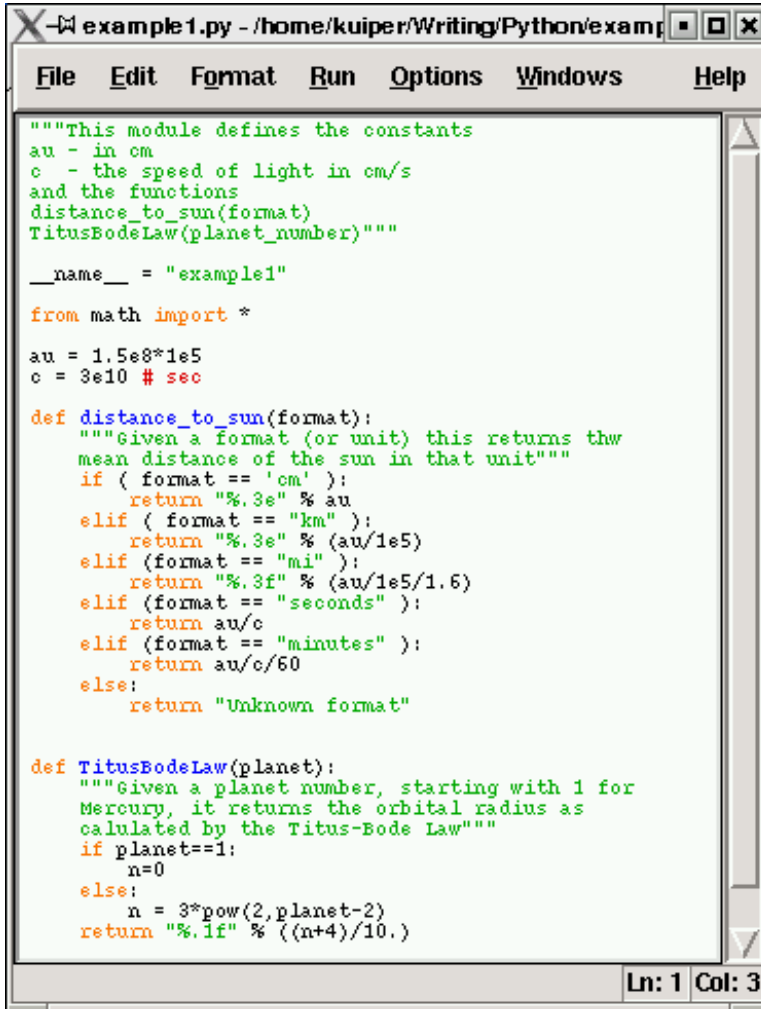
begins to make some sense. Remember that you can always try something quickly at the command prompt.

4 Help

There is, as part of the documentation of the Python distribution, a Global Module Index. That would be the place to start looking for a module name that suggests that it may have what you need for some task.

Clicking on `datetime` brings up the Python Library Reference page on this module. On a Linux system it would be found at `/usr/share/doc/python/html/lib/module-datetime.html`. The information here is definitely clearer than what the function `help()` provides.

For your own code, it is certainly less effort to provide good help information. Figure 3 shows the module `example1` with `help` documentation added. Your



```
"""This module defines the constants
au - in cm
c - the speed of light in cm/s
and the functions
distance_to_sun(format)
TitusBodeLaw(planet_number)"""

__name__ = "example1"

from math import *

au = 1.5e8*1e5
c = 3e10 # sec

def distance_to_sun(format):
    """Given a format (or unit) this returns the
    mean distance of the sun in that unit"""
    if (format == 'cm'):
        return "%.3e" % au
    elif (format == "km"):
        return "%.3e" % (au/1e5)
    elif (format == "mi"):
        return "%.3f" % (au/1e5/1.6)
    elif (format == "seconds"):
        return au/c
    elif (format == "minutes"):
        return au/c/60
    else:
        return "Unknown format"

def TitusBodeLaw(planet):
    """Given a planet number, starting with 1 for
    Mercury, it returns the orbital radius as
    calculated by the Titus-Bode Law"""
    if planet==1:
        n=0
    else:
        n = 3*pow(2,planet-2)
    return "%.1f" % ((n+4)/10.)
```

Figure 3: The module shown in Figure 1 with documentation added.

documentation will now be available.

Help on module `example1`:

```

NAME
    example1

FILE
    /home/kuiper/Writing/Python/example1.py

DESCRIPTION
    This module defines the constants
    au - in cm
    c - the speed of light in cm/s
    and the functions
    distance_to_sun(format)
    TitusBodeLaw(planet_number)

FUNCTIONS
    TitusBodeLaw(planet)
        Given a planet number, starting with 1 for
        Mercury, it returns the orbital radius as
        calculated by the Titus-Bode Law

    acos(...)
        acos(x)

        Return the arc cosine (measured in radians) of x.
....
    distance_to_sun(format)
        Given a format (or unit) this returns the
        mean distance of the sun in that unit
....
DATA
    au = 15000000000000.0
    c = 30000000000.0
    e = 2.7182818284590451
    pi = 3.1415926535897931

>>> help(example1.distance_to_sun)
Help on function distance_to_sun in module example1:

distance_to_sun(format)
    Given a format (or unit) this returns the
    mean distance of the sun in that unit

Notice that the math functions are part of example1 because of the form of the
import used. In the case of the math package, this is generally safe since the
function names are almost universally reserved for mathematical operations.

```

5 Organizing Code

5.1 Nesting Modules

Back to your own programming. Let's imagine that you have the module `jplephem` from Rick Fisher at NRAO. You compiled a module `coco` from FORTRAN sources by Wallace of the Starlink project using `pyfort`, and you have some of your own Python astronomy procedures in a module `astro`. You can include everything in `jplephem` and `coco` as part of your module `astro` by putting in the latter

```
import jplephem
import coco
```

If you import `astro` and do a `dir` you will see the contents of all the modules, as well as some others such as `math` and `Numeric`.

5.2 Packages

Packages are collections of modules which are arranged hierarchically like a file system. If a sub-directory below the top directory of the package contains a file `__init__.py`, which may be empty, then the contents of that sub-directory are available on the Python path in a dot-separated format:

```
import my_package.sub_dir1.sub_dir11.some_module as SM
```

A Module Gnuplot

The module `Gnuplot` provides an interface to the `gnuplot` program, which must be installed on the system. If you are not familiar with `Gnuplot` then skip this and look up the module `matplotlib`. It is much more powerful, as can be seen at <http://matplotlib.org/gallery.html>.

A.1 Class Gnuplot

A plotting session (there can be several simultaneously) is an instance of the class `Gnuplot`. The same name is used for the module and the class. For this reason, do not use `from Gnuplot import *`. Here is an example:

```
>>> import Gnuplot
>>> g1 = Gnuplot.Gnuplot()
>>> g2 = Gnuplot.Gnuplot()
```

We now have two plot objects. Any valid `gnuplot` command can be passed to such objects.

```
>>> g1('set xrange [-100:100])
>>> g1('set pointsize 2')
```

The objects can have attributes, such as a title.

```
g1.title('Graph 1')
```

A.2 Class PlotItem

A `Gnuplot` object knows how to plot one or more `PlotItem` objects. There are methods to create various types of such objects.

```
>>> import Numeric
>>> x = arange(10, typecode='fd')
>>> y1 = x**2
>>> p1 = Gnuplot.Data(x,y1,title='x^2')
>>> p2 = Gnuplot.Func('x**2')
>>> p3 = Gnuplot.Data([[0.5,0.25],[1.5,2.25],[2.5,6.25],\
                      [3.5,12.25]],title='more data')
>>> p4 = Gnuplot.File('gnuplot_ex1.dat')
>>> g.xlabel('x')
>>> g.ylabel('y')
>>> g.plot(p1,p2,p3,p4)
```

There is one other type of `PlotItem`, which is generated by function `GridData`. The arguments are, in this order, a two-dimensional data array with dimensions (N_x, N_y) , an one-dimensional array of corresponding x -values, and a similar array of y -values. There are other, optional arguments identified with keywords, like the keyword `title` in the above example.

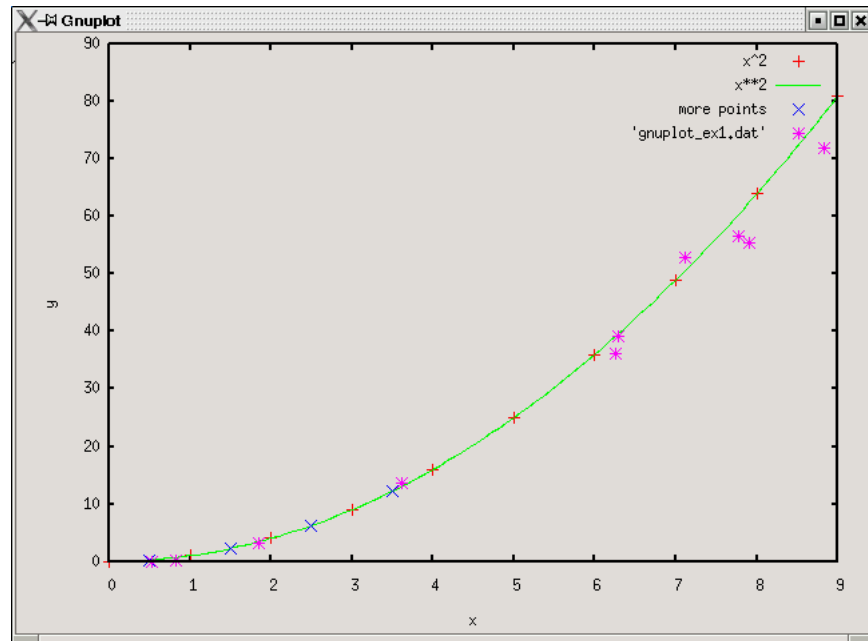


Figure 4: An example of the use of the **Gnuplot** module. See Appendix B for how the data in the file were generated.

B Module random

This illustrates the use of the **random**.

```
#!/usr/bin/python

import math
import random

for i in range(10):
    x = 9*random.random()
    y = pow(x+random.gauss(0,0.3),2)
    print x,y
```

It also shows how to create an executable Python script. The data file was created with

```
./gnuplot_ex1a.py > gnuplot_ex1.dat
```

B.1 Functions

These are the same as the methods of the class `Random` (see below). Basically, they can be invoked without creating an object.

```
>>> random.randint(9,19)
15
```

For people who want to give object-oriented programming as wide a berth as possible, this may be the best way to go. The documentation obtained from

```
>>> help(random)
```

gives all the functions after the definitions of the classes, and is pretty clear. Ignore the first argument `self`. Arguments with defaults show the default value after an equal sign.

B.2 Classes

B.2.1 Class `Random`

This is the base random number generator with, among others, the following methods:

`random()` Return a float in the interval (0, 1).

`seed(<optional>)` Initialize internal state from hashable object.

`gauss(mean, standard_deviation)`

`randint(a, b)` Return random integer in range [a, b], including both end points.

`randrange(start, stop=None, step=1)` Choose a random item from `range(start, stop[, step])`. In this case, following the Python convention, the endpoint is not included.

`uniform(a, b)` Get a random number in the range [a, b).

and many more. The documentation obtained from

```
>>> help(random)
```

gives all the methods and is pretty clear.

B.2.2 Class `WichmannHill(Random)`

This is a subclass of `Random`. It's a different way of generating random numbers. It has almost all the same methods as `Random`.