Math 3040: Introduction to Python

M. M. Sussman

`sussmanm@math.pitt.edu`

Office Hours: M-Th 11:10-12:10, Thack 622

May 12, 2014

Contents

Introduction to Python

Resources I have used in preparing this introduction.

1. **`http://www.stavros.io/tutorials/python/`**
2. **`https://docs.python.org/2/tutorial/`**

Getting help

- From the command line: **`pydoc`** *object name*
- From the Python prompt: **`help(`***object name***`)`**

Contents

Running python

- **The best way is to use `spyder`**
- Can run **`python`** or **`ipython`** from a command prompt
- “Applications” $\rightarrow$ “Development” $\rightarrow$ **`idle`** (using Python 2.7)
- Can run **`idle`** from a command prompt
- Can run **`ipython notebook`** (browser-based “notebook” interface similar to Mathematica)

Spyder

1. K Menu→Applications→Development→Spyder
2. “Spyder is a powerful interactive development environment“
3. Editing
4. Interactive testing and debugging
5. Introspection
6. Aimed toward the scientific commjunity
7. Open source, running on Linux, Mac, MS-Windows

Spyder demo

1. Open IPython console
2. Automatic “pylab”
   ```
   from numpy import *
   from scipy import *
   from matplotlib import *
   ```
3. Save console using Ctrl-S
   - Can be used as part of your homework submission

Running python with a file without Spyder

- Filename *should* have **`.py`** extension.
- **`python`** *filename.py* from a command prompt

Contents

File structure and line syntax

- No mandatory statement termination character.
- Blocks are determined by indentation
- Statements requiring a following block end with a colon (`:`)
- Comments start with octothorpe (`#`), end at end of line
- Multiline comments are surrounded by triple double quotes (`"""`) or triple single quotes (`'''`)
- Continue lines with `\`

```
"""
Example of a file with blocks in it
example1.py
"""
print "Hello!"                  # just print a string
x=input("Guess an integer ")    # dangerous function!
if x > 10:                      # colon
    print "A big number :-)"
blank line when typing
else:                           # colon
    print "Not big enough :-("
blank line when typing
```

Debugging hint (`'''`)

One strategy during debugging:

1. Add special-purpose code
2. Test corrected code
3. “Comment out” the special-purpose code instead of removing it at first
4. Triple *single* quotes are good
5. Easy to find for later cleanup

Formatted printing

Format controls as in C++, MATLAB, *etc.*

```
>>> n=35
>>> e=.00114
>>> print "Step %d, error=%e"%(n,e)
Step 35, error=1.140000e-03

>>> print "Step %d, error=%f"%(n,e)
Step 35, error=0.001140

>>> print "Step %d, error=%11.3e"%(n,e)
Step 35, error=  1.140e-03
```

Contents

Python basic data types

- Integers: `0, -5, 100`
- Floating-point numbers: `3.14159, 6.02e23`
- Complex numbers: `1.5 + 0.5j`
- Strings: `"A string"` or `'another string'`
  - Stick to double-quotes

Python basic data types

- Integers: `0, -5, 100`
- Floating-point numbers: `3.14159, 6.02e23`
- Complex numbers: `1.5 + 0.5j`
- Strings: `"A string"` or `'another string'`
  - Stick to double-quotes
- Unicode strings: `u"A unicode string"`

Python basic data types

- Integers: `0, -5, 100`
- Floating-point numbers: `3.14159, 6.02e23`
- Complex numbers: `1.5 + 0.5j`
- Strings: `"A string"` or `'another string'`
  - Stick to double-quotes
- Unicode strings: `u"A unicode string"`
- Logical or Boolean: `True, False`
- `None`

Basic operations

- `+, -, *, /`
- `**` (raise to power)
- `//` ("floor" division)
- `%` (remainder)
- `divmod, pow`
- `and, or, not`
- `>=, <=, ==, !=` (logical comparison)

```
>>> x=10
>>> 3*x
30
>>> x-2
8
>>> x/3
3
>>> x>5
True
>>> divmod(x,3)
(3, 1)
>>> pow(x,3)
1000
```

Python array-type data types

- Numerical array data type is in numpy (later)

Python array-type data types

- Numerical array data type is in numpy (later)
- List: `[0,"string",`*another list* `]`

Python array-type data types

- Numerical array data type is in numpy (later)
- List: `[0,"string",`*another list* `]`
- Tuple: immutable list, surrounded by `()`

Python array-type data types

- Numerical array data type is in numpy (later)
- List: `[0,"string",`*another list* `]`
- Tuple: immutable list, surrounded by `()`
- Dictionary (dict): `{"key1":"value1",2:3,"pi":3.14}`

Data types have “attributes”

- Lists have only function attributes. If **`L`** is a list, then
  1. **`L.append(x)`** appends x to the list
  2. **`L.index(x)`** finds the first occurrance of x in the list
  3. **`x=L.pop()`** return last item on list and remove it from list

Equals, Copies, and Deep Copies

```
>>> import copy
```

Equals, Copies, and Deep Copies

```
>>> import copy

>>> x=[1,2]
>>> y=[3,4,x]
>>> z=y
>>> print x,y,z
[1, 2] [3, 4, [1, 2]] [3, 4, [1, 2]]
```

Equals, Copies, and Deep Copies

```
>>> import copy

>>> x=[1,2]
>>> y=[3,4,x]
>>> z=y
>>> print x,y,z
[1, 2] [3, 4, [1, 2]] [3, 4, [1, 2]]

>>> c=copy.copy(y)
>>> d=copy.deepcopy(y)
>>> print "y=",y," z=",z," c=",c," d=",d
y= [3, 4, [1, 2]]  z= [3, 4, [1, 2]]  c= [3, 4, [1, 2]]  d= [3, 4, [1, 2]]
```

Equals, Copies, and Deep Copies

```
>>> import copy

>>> x=[1,2]
>>> y=[3,4,x]
>>> z=y
>>> print x,y,z
[1, 2] [3, 4, [1, 2]] [3, 4, [1, 2]]

>>> c=copy.copy(y)
>>> d=copy.deepcopy(y)
>>> print "y=",y," z=",z," c=",c," d=",d
y= [3, 4, [1, 2]]  z= [3, 4, [1, 2]]  c= [3, 4, [1, 2]]  d= [3, 4, [1, 2]]

>>> y[0]="*"
>>> print "y=",y," z=",z," c=",c," d=",d
```

Equals, Copies, and Deep Copies

```
>>> import copy

>>> x=[1,2]
>>> y=[3,4,x]
>>> z=y
>>> print x,y,z
[1, 2] [3, 4, [1, 2]] [3, 4, [1, 2]]

>>> c=copy.copy(y)
>>> d=copy.deepcopy(y)
>>> print "y=",y," z=",z," c=",c," d=",d
y= [3, 4, [1, 2]]  z= [3, 4, [1, 2]]  c= [3, 4, [1, 2]]  d= [3, 4, [1, 2]]

>>> y[0]="*"
>>> print "y=",y," z=",z," c=",c," d=",d
y= ["*", 4, [1, 2]]  z= ["*", 4, [1, 2]]  c= [3, 4, [1, 2]]  d= [3, 4, [1, 2]]
```

Equals, Copies, and Deep Copies

```
>>> import copy

>>> x=[1,2]
>>> y=[3,4,x]
>>> z=y
>>> print x,y,z
[1, 2] [3, 4, [1, 2]] [3, 4, [1, 2]]

>>> c=copy.copy(y)
>>> d=copy.deepcopy(y)
>>> print "y=",y," z=",z," c=",c," d=",d
y= [3, 4, [1, 2]]  z= [3, 4, [1, 2]]  c= [3, 4, [1, 2]]  d= [3, 4, [1, 2]]

>>> y[0]="*"
>>> print "y=",y," z=",z," c=",c," d=",d
y= ["*", 4, [1, 2]]  z= ["*", 4, [1, 2]]  c= [3, 4, [1, 2]]  d= [3, 4, [1, 2]]

>>> z[2][0]=9
>>> print "y=",y," z=",z," c=",c," d=",d
```

Equals, Copies, and Deep Copies

```
>>> import copy

>>> x=[1,2]
>>> y=[3,4,x]
>>> z=y
>>> print x,y,z
[1, 2] [3, 4, [1, 2]] [3, 4, [1, 2]]

>>> c=copy.copy(y)
>>> d=copy.deepcopy(y)
>>> print "y=",y," z=",z," c=",c," d=",d
y= [3, 4, [1, 2]]  z= [3, 4, [1, 2]]  c= [3, 4, [1, 2]]  d= [3, 4, [1, 2]]

>>> y[0]="*"
>>> print "y=",y," z=",z," c=",c," d=",d
y= ["*", 4, [1, 2]]  z= ["*", 4, [1, 2]]  c= [3, 4, [1, 2]]  d= [3, 4, [1, 2]]

>>> z[2][0]=9
>>> print "y=",y," z=",z," c=",c," d=",d
y= ["*", 4, [9, 2]]  z= ["*", 4, [9, 2]]  c= [3, 4, [9, 2]]  d= [3, 4, [1, 2]]

>>> x
[9, 2]
```

Equals, Copies, and Deep Copies

```
>>> import copy

>>> x=[1,2]
>>> y=[3,4,x]
>>> z=y
>>> print x,y,z
[1, 2] [3, 4, [1, 2]] [3, 4, [1, 2]]

>>> c=copy.copy(y)
>>> d=copy.deepcopy(y)
>>> print "y=",y," z=",z," c=",c," d=",d
y= [3, 4, [1, 2]]  z= [3, 4, [1, 2]]  c= [3, 4, [1, 2]]  d= [3, 4, [1, 2]]

>>> y[0]="*"
>>> print "y=",y," z=",z," c=",c," d=",d
y= ["*", 4, [1, 2]]  z= ["*", 4, [1, 2]]  c= [3, 4, [1, 2]]  d= [3, 4, [1, 2]]

>>> z[2][0]=9
>>> print "y=",y," z=",z," c=",c," d=",d
y= ["*", 4, [9, 2]]  z= ["*", 4, [9, 2]]  c= [3, 4, [9, 2]]  d= [3, 4, [1, 2]]

>>> x
[9, 2]
```

Moral: Only **deepcopy** does it right!

pydoc for help

`$ pydoc list` OR `>>> help (list)`

```
Help on class list in module __builtin__:

class list(object)
 |  list() -> new empty list
 |  list(iterable) -> new list initialized from iterable's items
 |
 |  Methods defined here:
 |  __add__(...)
 |      x.__add__(y) <==> x+y
 |  __contains__(...)
 |      x.__contains__(y) <==> y in x
 |                more to ignore
 |  append(...)
 |      L.append(object) - append object to end
 |  count(...)
 |      L.count(value) -> integer - return number of occurrences of value
 |  extend(...)
 |      L.extend(iterable) - extend list by appending elements from the iterable
 |  index(...)
 |      L.index(value, [start, [stop]]) -> integer - return first index of value
 |      Raises ValueError if the value is not present.
 |  insert(...)
 |  pop(...)
 |      L.pop([index]) -> item - remove and return item at index (default last).
 |      Raises IndexError if list is empty or index is out of range.
 |  remove(...)
 |      L.remove(value) - remove first occurrence of value.
 |      Raises ValueError if the value is not present.
```

Flow control

- **`if`**
- **`for`**
- **`while`**
- **`range(N)`** generates the numbers 0,. . . ,N

```
# print out even numbers
for n in range(13):
    if n%2 == 0:
        print n
    else:
        # not necessary
        continue
```

Assert

One *extremely* valuable feature of Python is the **`assert`**.

- Use it whenever you think something is impossible!
- "Impossible" branches of if-tests
- "Impossible" endings of loops
- You will be expected to use **`assert`**!

```
if x > 0:
    some code for positive x
elseif x < 0:
    some code for negative x
else:
    # x should never to be zero!
    assert(x!=0)
```

Functions

- Functions begin with **`def`**
- The **`def`** line ends with a colon
- Functions use **`return`** to return values

Functions

- Functions begin with **`def`**
- The **`def`** line ends with a colon
- Functions use **`return`** to return values

```
def sine(x):
    """
    compute sin(x) to error of 1.e-10
    using Maclaurin (Taylor) series
    """
```

Functions

- Functions begin with **`def`**
- The **`def`** line ends with a colon
- Functions use **`return`** to return values

```
def sine(x):
    """
    compute sin(x) to error of 1.e-10
    using Maclaurin (Taylor) series
    """
    tol=1.e-10
    s=x
    t=x
    n=1
    while abs(t) > tol:  # abs is built-in
        n+=2
        t=(-t)*x*x/(n*(n-1))
        s+=t
```

Functions

- Functions begin with **def**
- The **def** line ends with a colon
- Functions use **return** to return values

```
def sine(x):
    """
    compute sin(x) to error of 1.e-10
    using Maclaurin (Taylor) series
    """
    tol=1.e-10
    s=x
    t=x
    n=1
    while abs(t) > tol:  # abs is built-in
        n+=2
        t=(-t)*x*x/(n*(n-1))
        s+=t
        assert(n<10000) # too long!  Do something else!
    return s
```

Functions

- Functions begin with **def**
- The **def** line ends with a colon
- Functions use **return** to return values

```
def sine(x):
    """
    compute sin(x) to error of 1.e-10
    using Maclaurin (Taylor) series
    """
    tol=1.e-10
    s=x
    t=x
    n=1
    while abs(t) > tol:  # abs is built-in
        n+=2
        t=(-t)*x*x/(n*(n-1))
        s+=t
        assert(n<10000) # too long!  Do something else!
    return s
```

Importing and naming

- Include external libraries using **`import`**
- **`import numpy`**
  Imports all numpy functions, call as **`numpy.sin(x)`**
- **`import numpy as np`**
  Imports all numpy functions, call as **`np.sin(x)`**
- **`from numpy import *`**
  Imports all numpy functions, call as **`sin(x)`**
- **`from numpy import sin`**
  Imports only **`sin()`**

Pylab in Spyder

Automatically does following imports

```
from pylab import *
from numpy import *
from scipy import *
```

You *must* do your own importing when writing code in files

I strongly suggest using correct names.

```
import numpy as np
import scipy.linalg as la
import matplotlib.pyplot as plt
```

Contents

A Class is a generalized data type

- **`numpy`** defines a class called **`ndarray`**
- Define variable **`x`** of type **`ndarray`**, a one-dimensional array of length 10:
  ```
  import numpy as np
  x=np.ndarray([10])
  ```
- Varibles of type **`ndarray`** are usually just called “array”.

Classes define members’ “attributes”

- Attributes can be data
  - Usually, data attributes are “hidden”
  - Names start with double-underscore
  - Programmers are trusted not to access such data
- Attributes can be functions
  - Functions are provided to access “hidden” data

Examples of attributes

One way to generate a **`numpy`** array is:

```
import numpy as np
x=np.array([0,0.1,0.2,0.4,0.9,3.14])
```

- (data attribute) **`x.size`** is **`6`**.
- (data attribute) **`x.dtype`** is **`"float64"`** (quotes mean “string”)
- (function attribute) **`x.item(2)`** is **`0.2`** (parentheses mean “function”)

Operators can be overridden

- Multiplication and division are pre-defined (overridden)

```
>>> 3*x
array([ 0.   ,  0.3 ,  0.6 ,  1.2 ,  2.7 ,  9.42])
```

- Brackets can be overridden to make things look “normal”

```
>>> x[2] # bracket overridden
0.2
```

Inheritance

- Suppose you write a program about ellipses.
- You “abstract” an ellipse as a plane figure with major and minor axes.
- You use its area and its circumference, but nothing else.

Inheritance

- Suppose you write a program about ellipses.
- You “abstract” an ellipse as a plane figure with major and minor axes.
- You use its area and its circumference, but nothing else.
- Someone comes by and asks you to apply your program to circles.

Inheritance

- Suppose you write a program about ellipses.
- You "abstract" an ellipse as a plane figure with major and minor axes.
- You use its area and its circumference, but nothing else.
- Someone comes by and asks you to apply your program to circles.
- You could just say, "Define your circle as an ellipse with major and minor axes equal" (problem solved)

Inheritance

- Suppose you write a program about ellipses.
- You “abstract” an ellipse as a plane figure with major and minor axes.
- You use its area and its circumference, but nothing else.
- Someone comes by and asks you to apply your program to circles.
- You could just say, “Define your circle as an ellipse with major and minor axes equal” (problem solved)
- Awkward, mistake-prone, and unfriendly

Inheritance

- Suppose you write a program about ellipses.
- You “abstract” an ellipse as a plane figure with major and minor axes.
- You use its area and its circumference, but nothing else.
- Someone comes by and asks you to apply your program to circles.
- You could just say, “Define your circle as an ellipse with major and minor axes equal” (problem solved)
- Awkward, mistake-prone, and unfriendly
- Define a circle that IS an ellipse but with major and minor axes forced to be equal.

Inheritance

- Suppose you write a program about ellipses.
- You “abstract” an ellipse as a plane figure with major and minor axes.
- You use its area and its circumference, but nothing else.
- Someone comes by and asks you to apply your program to circles.
- You could just say, “Define your circle as an ellipse with major and minor axes equal” (problem solved)
- Awkward, mistake-prone, and unfriendly
- Define a circle that IS an ellipse but with major and minor axes forced to be equal.
- Don’t have to write much code!
- Can use it wherever an ellipse was used before!
- Don’t have to debug stuff you are reusing.

Inheritance II

- Someone comes by and asks you to apply your program to rectangles
- Still have area and circumference.

Inheritance II

- Someone comes by and asks you to apply your program to rectangles
- Still have area and circumference.
- Define a rectangle that IS an ellipse, but with modified area and circumference functions.

Inheritance II

- Someone comes by and asks you to apply your program to rectangles
- Still have area and circumference.
- Define a rectangle that IS an ellipse, but with modified area and circumference functions.
- Lots of new code, but downstream code does not change!