Python inspired by problem set 3

Jarrod Millman
Adapted by
Harold Pimentel
Statistics 243
UC Berkeley

November 2, 2015

In groups of two to three, I want you to spend about 10 minutes on each of the following questions. For most of the questions, you will need to use IPython to try out the code snippets. While you are discussing things, I will circulate among the groups to answer questions and observe. After working on each question for 10 minutes in small groups, we will have a group discussion for about 10 minutes. And so on.

You may encounter aspects of Python that you haven't seen before. Hopefully, you will be able to get some sense of what the code snippet does by trying it out with IPython. If you can't get something working from IPython, please ask me for clarification or help.

1. Scope

   You can add an attribute to a function at any point during runtime. As long as the attribute is assigned a value prior to being needed, everything works. If you try to use an attribute before it is created (through assignment), Python will raise an `AttributeError`.

```
def myfunc(x):
    if not x in myfunc.table:
        myfunc.table[x] = x*10
    return myfunc.table[x]
```

   If you call the function now, you should get the following error message:[1]

```
AttributeError: 'function' object has no attribute 'table'
```

   That error means that `myfunc.table` has not been created yet. To create it just assign it a value. In this case you will want to create an empty dictionary.

```
myfunc.table = {}
```

   Alternatively, you could initially populate it with some already known or precalculated entries. For example, you could save the resulting dictionary from an early run of your code to disk. Then when you want to use the code again, you could load the saved dictionary from disk.[2]

   Now use `type(myfunc)` and `type(myfunc.table)` to examine the type of object that `myfunc` and `myfunc.table` are labels for. Check that `myfunc.table` is still an empty dictionary. Now call `myfunc()` on an argument (e.g., 3) and look to see what happens to `myfunc.table`. Try calling `myfunc()` with different types of

[1] Sorry about the curly quotes, `knitr`'s Python engine doesn't properly handle single quotes. You will notice that I use double quotes in the Python snippets. Normally, I would use single quotes in many instances as it is less obtrusive and doesn't require the use of the `Shift` key.

[2] If you need to save your dictionary to disk, you may want to use `JSON`. `JSON` is an open standard, human-readable text file format that is widely used as an alternative the `XML`. And is one of the most popular methods for exchanging data on the web.

arguments (e.g., give it a string or a list). Do you get an error message? Look at `myfunc.table`. Did it change? If so, what happened and why?

This is an example of how you could use Python to memoize a function.[3] Memoization is a way to speed up a function by caching results once they've been computed, so that you don't have to compute them again. Can you think of cases where this would be useful?

2. Memory

As I mentioned in an earlier section, variables are not their values in Python (think "I am not my name, I am the person named XXX"). Certain objects in Python are mutable (e.g., lists, dictionaries), while other objects are immutable (e.g., tuples, strings). Many objects can be composite (e.g., a list of dictionaries or a dictionary of lists, tuples, and strings). For this example, you are going to explore mutable and immutable objects by examining how they compose.

First, create a list, a tuple, and a dictionary containing the list and the tuple as values.

```
l1 = [1, 2]
t1 = (3, 4)
d = {"a": l1, "b": t1}
```

Look at the objects you've created.[4] Check their types. Use the `id` function to find the memory addresses of the various objects. For instance, you might try something like this:

```
d
type(d)
type(d["a"])
id(d)
```

What happens to `l1` when you modify `d["a"][1]`? What about when you modify `d["a"]`? You might try:

```
id(l1)
l1
id(d["a"])
d["a"][1] = 1
d["a"]
l1
id(l1)
id(d["a"])
```

Similarly, what happens when you modify `d["b"][1]`? What about when you modify `d["b"]`? You might try:

```
d["b"][1] = 1
d["b"] = 1
```

Now try to make a copy of `d`. First, create a new variable by assign the old variable to it.

```
e = d
e is d
```

What happened? If you use tab-completion on `d.` from IPython, you will see it has a `copy` method.

[3] Python 3 provides a decorator to simplify this process:
`https://docs.python.org/3/library/functools.html#functools.lru_cache`

[4] Make sure you see which object is composite. If you have time, you may wish to examine whether you can further nest objects. Can you create a dictionary with a list of dictionaries of lists?

```
e = d.copy()
e is d
e["a"] is d["a"]
```

What does that do? Finally make a "deepcopy" of `d`.

```
from copy import deepcopy
e = deepcopy(d)
e is d
e["a"] is d["a"]
```

Can you explain the differences between the various ways the above copies work?

3. Random walks

For question 4, you were asked to implement a random walk function. As a reminder, here is the example solution.

```
makeWalk <- function(n, fullpath = TRUE){
  # using cumsum() to avoid loops
  if(length(n) > 1 || !is.numeric(n) ||  n %% 1 != 0 || n < 1)
    stop("rw_fast: 'n' must be a positive integer")
  updown <- sample(c(0, 1), n, replace = TRUE)
  steps <-  sample(c(-1, 1), n, replace = TRUE)
  xsteps <- (updown == 0) * steps
  ysteps <- (updown == 1) * steps
  x <- c(0, cumsum(xsteps))
  y <- c(0, cumsum(ysteps))
  if(fullpath){
    return(cbind(x, y))
  } else{
    return(c(x[n+1], y[n+1]))
  }
}
```

Here is an implementation in Python.

```
import numpy as np
import matplotlib.pyplot as plt

np.random.seed(2)

code  = {"up": (0,1),
         "down": (0,-1),
         "left": (-1,0),
         "right": (1,0)}

def random_2d_walk(nsteps=100):
    steps = np.random.choice(code.keys(), nsteps)
    walk = np.array([code[step] for step in steps])
    xy = walk.cumsum(axis=0)
    return (xy, walk)

xy, walk = random_2d_walk()
plt.plot(xy[:,0], xy[:,1])
# plt.savefig("test.png")
```

Compare the two versions. Try to find parallels in the implementation (I tried to mimic the R implementation mostly so hopefully they look somewhat similar to you). Do you prefer one over the other? How about specific code snippets.

Note the syntax used to create the variable `walk` in the definition of the `random_2d_walk()` function. The argument to the `np.array()` constructor is created using list comprehension. This notation should remind you of the mathematical set-builder notation.[5]

In the Python implementation, I am not checking the type and value of my input. If I wanted to so, I could something like this:

```
def random_2d_walk(nsteps=100):
    if not isinstance(nsteps, int) or nsteps < 1:
        raise ValueError("Number of steps must be a positive integer.")
    ...
```

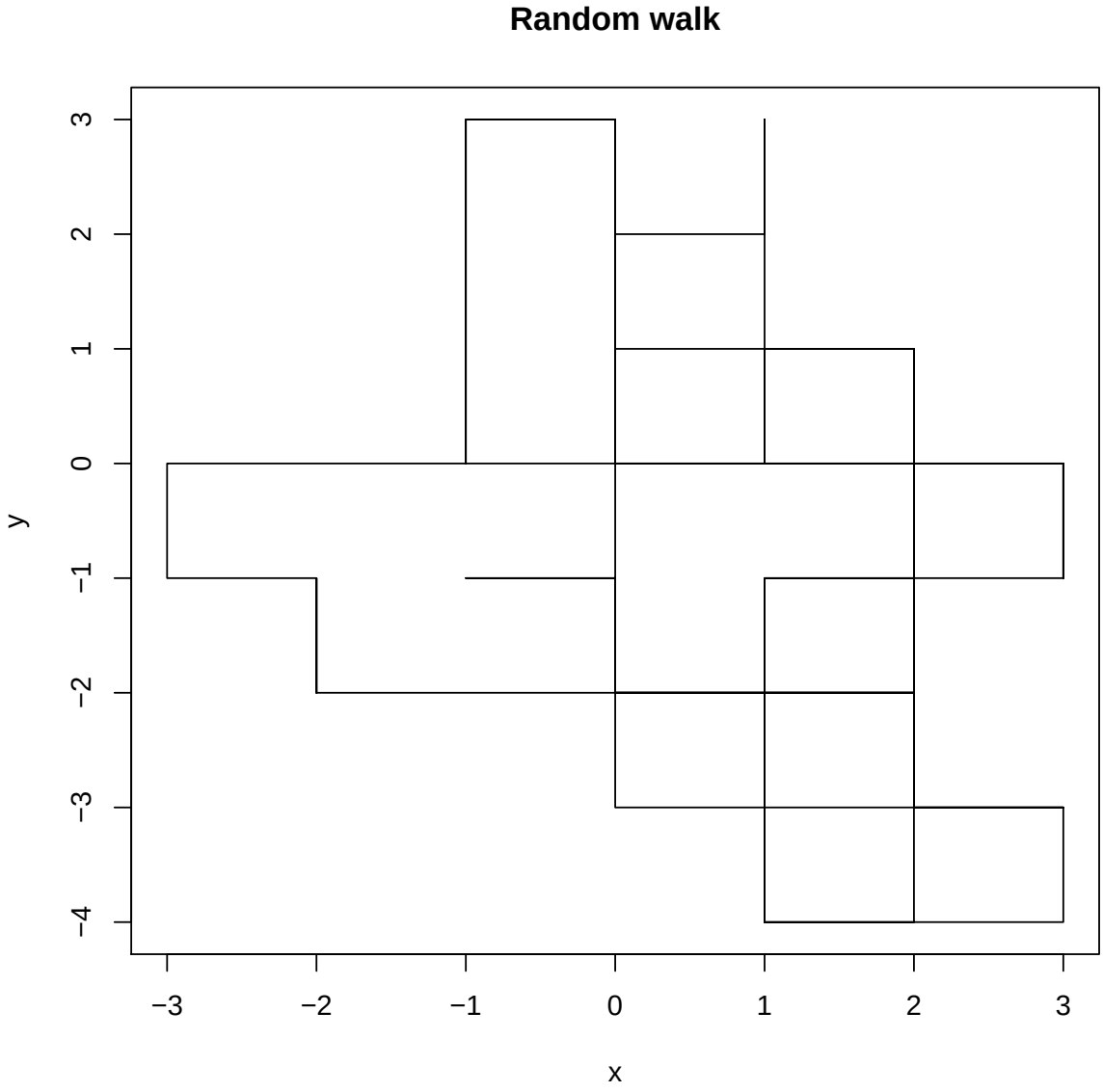
It is generally not consider Pythonic to check argument types explicitly. The focus is on whether the interface is implemented. This is called *duck typing*. It is normally considered desirable to let the function error if the input is invalid. It you want to handle the error more directly, then a common approach is to try using the input and then handle any exceptions explicitly.

Here is the example solution for creating an S3 class in R.

```
rw <- function(n = 100, start = c(0, 0), fullpath = TRUE){

  path <- makeWalk(n, fullpath)
  if(fullpath){
    if(!identical(start, c(0, 0)))
      path <- t(t(path) + start)
  } else{ # don't bother to check if start is non-zero
          #   as time to add is trivial
    path <- path + start
  }

  obj <- list(path = path, length = n+1)
  class(obj) <- "rw"
  return(obj)
}

print.rw <- function(obj){
  cat("Random walk of length ", obj$length, ".\n", sep = "")
  invisible(obj) # following template of print.lm()
}

# user can pass in additional args through the ... mechanism
plot.rw <- function(obj, type = "l", main = "Random walk", xlab = "x", ylab = "y", ...){
  plot(x = obj$path[ , 1], y = obj$path[ , 2], main = main,
       xlab = xlab, ylab = ylab, type = type, ...)
}

`[.rw` <- function(obj, i){
  return(obj$path[i, ])
}

# generic start replacement function
```

[5] Creating sets using the set-builder notation is known as set comprehension.

```r
`start<-` <- function(x, ...){
  UseMethod("start<-")
}

`start<-.rw` <- function(obj, value){
  if(is.numeric(value) && length(value) == 2 && sum(value%%1) == 0){
    obj$path <- t(t(obj$path) - obj$path[1] + value)
    # note use of [.rw operator
  } else{
    stop("Value argument needs to be a vector of integers of length 2")
  }
  return(obj)
}

out <- rw(100)
out

## Random walk of length 101.

out[5]

##  x  y
##  2  0

plot(out)
```


Random walk
y
x
3
2
1
0
−1
−2
−3
−4
−3
−2
−1
0
1
2
3

Here is an object-oriented implementation in Python.[6]

```
import numpy as np
import matplotlib.pyplot as plt

class Random2DWalk(object):

    def __init__(self, start=(0,0)):
        self.steps = ["start"]
        self.walk = np.array([start])
        self._code = {"up": (0,1),
                      "down": (0,-1),
                      "left": (-1,0),
                      "right": (1,0)}

    def __str__(self):
        current_position = self.position()[-1]
        total_steps = len(self.steps) - 1
        message = "After {0} steps you are at position: {1}"
        return message.format(total_steps, current_position)

    def step(self, nsteps=100):
        steps = np.random.choice(self._code.keys(), nsteps)
        walk = np.array([self._code[step] for step in steps])
        self.steps += steps
        self.walk = np.vstack([self.walk, walk])
        return None

    def position(self):
        return self.walk.cumsum(axis=0)

    def plot(self):
        xy = self.position()
        plt.plot(xy[:,0], xy[:,1])
        plt.savefig("test.png")
        return None

np.random.seed(2)
rw = Random2DWalk()
print rw
rw.step()
print rw
print rw.steps[:5]
print rw.walk[:5]
rw.plot()

## After 0 steps you are at position: [0 0]
## After 100 steps you are at position: [-9  7]
## ['start', 'down', 'left', 'right', 'down']
## [[ 0  0]
##  [ 0 -1]
##  [-1  0]
##  [ 1  0]
##  [ 0 -1]]
```

[6] In order to make the `position` method act like an attribute, you could use the `@property` decorator.

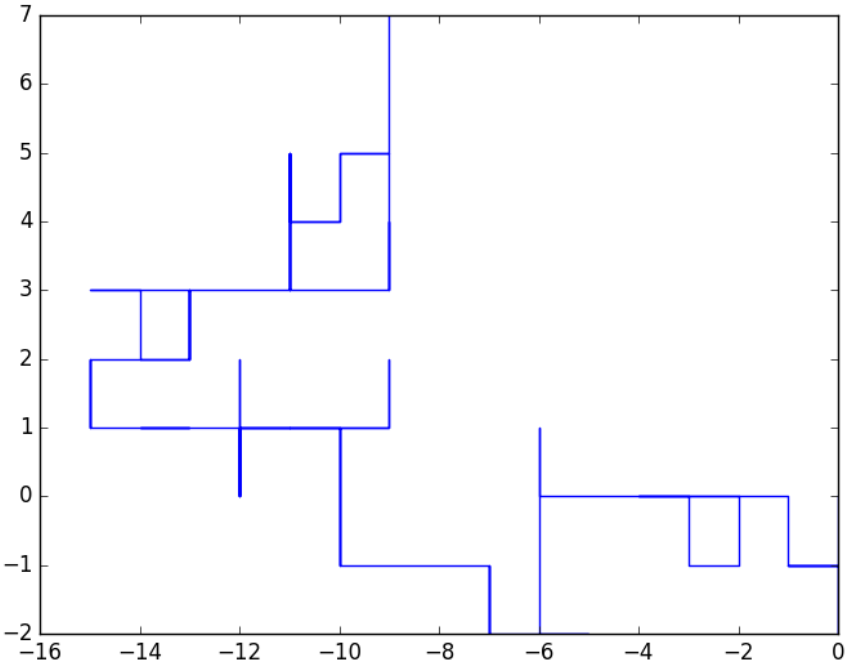


Figure 1: Plot of a random walk on a 2-dimensional grid.

Since `knitr` is not able to display figures generated by Python, I've included the figure generated by the above code in Figure 1. I've also used the `print` statement to include the output. If you are working at an interactive Python prompt, you can omit it. You also might prefer to use `plt.show()` rather than saving the figure to disk with `plt.savefig()`.

Look through this implementation and see if you can understand how it works. Note the use of `self`.[7] It is the first argument to the class methods and is how the methods are able to refer to the attributes of an instance.

Compare the two class implementations. Are there any major differences in how classes are written in Python and R? Is one of the implementations clearer or easy to read? Is that due to your familiarity with R? Are you able to work out what Python is doing? Does its syntax seem obscure? Type the class definition at the IPython prompt and create an instance of it. Use tab-completion to explore the object. Are you able to figure out how to access and work with its attributes and methods?

After considering the similarities and differences in how Python and R implement object-oriented programming, can you think of any design criteria that might explain the differences between how R and Python implement object-oriented programming? For instance, would one be easier to use on-the-fly while working interactively? Does one implementation seem like it would be better to use in a larger codebase? Why or why not?

4. Bonus 1

   R generally does not have pass-by-reference semantics, but you can emulate them in certain circumstances. Below is a function that modifies the argument passed in.

   - Explore how it behaves under different inputs.
   - How does the memory behave under different inputs? Hint: use `.Internal(inspect)`

[7] The use of `self` is a convention. You can use another name (e.g., `this`), but it is probably best to use the convention.

```
modify_argument <- function(x) {
  x$x <- 10
}

a_list <- list(x = 3)
an_environment <- new.env()
an_environment$x <- 3

modify_argument(a_list)
modify_argument(an_environment)
```

5. Bonus 2

   For problem set 5, you examined underflow when adding many small values to a large value. Normally you would use the `sum()` method of the `ndarray` rather than Python's builtin `sum()` function. If you wanted to use higher precision, then you would specify it explicitly.

```
import numpy as np

x = np.array([1e-16]*(10001))
x[0]=1
sum(x)
x.sum()
x.sum(dtype=np.float128)

from math import fsum
fsum(x)
```

   Note that `math.fsum` uses an alternative algorithm to ensure that it doesn't loose precision. Look at the doctring for `np.sum`. Can you see why specifying the `dtype` works?

   You may find the following code and discussion interesting.[8] Finally, for the interested, Python also supports rational arthimetic.[9]

   For a more details on floating-point arthimetic, please see the classic text by David Goldberg.[10]

[8] `http://code.activestate.com/recipes/393090/`
[9] `https://docs.python.org/2/library/fractions.html`
[10] `http://docs.oracle.com/cd/E19957-01/806-3568/ncg_goldberg.html`