Python for Finance

Control Flow, data structures and first application (part 2)

Andras Niedermayer

Dauphine Université Paris

Outline

Functions

- definition with parameters

```
In [1]: def hello(name):
...   :     print 'Hello␣' + name
```

- usage:

```
In [1]: hello('Alice')
In [2]: hello('Bob')
```

- even better: document your code with a docstring

```
In [1]: def hello(name):
...   :     """
...   :     A function that says hello
...   :     to `name`
...   :     """
...   :     print 'Hello␣' + name
In [2]: help(hello)
```

Functions

- return values

```
def get_answers(answer_number):
    if answer_number == 1:
        return 'You␣are␣the␣first.'
    elif answer_number == 2:
        return 'Twice␣as␣good.'
    else:
        return 'Something␣else␣than␣one'\
               '␣or␣two'
```

- usage:

```
In [1]: print get_answer(2)
```

Functions

- Or pick a random one:

```
import random
fortune = get_answer(random.randint(1,9))
print fortune
```

Variable Scope

- one cannot use variables outside of their scope

```
def spam():
    eggs = 31337
    print(str(eggs))
spam()
print(eggs)
```

Variable Scope

- This also true for calls to other functions:

```
def spam():
    eggs = 31337
    bacon()
    print(eggs)

def bacon():
    ham = 101
    eggs = 0

spam()
```

Variable Scope

- In constrast to this, one can use global variables in a funtion:

```python
def spam():
    print(eggs)

eggs = 31337
spam()
```

Functions

- The following is bad style, but feasible:

```
def spam():
    eggs = 'spam␣local'
    print(eggs)

def bacon():
    eggs = 'bacon␣local'
    print(eggs)
    spam()
    print(eggs)

eggs = 'global'
bacon()
```

Variable Scope

- Usage of the ‘global’ keyword

```
def spam():
    global eggs
    eggs = spam

eggs = 'global'
spam()
print(eggs)
```

Our Own Module

- Definition in Python documentation:
  "A module is a file containing Python definitions and statements. The file name is the module name with the suffix .py appended."
- A module is essentially a file that one imports. The file contains the functions we can call.
- Here is an example of what we may try to achieve:

```
In [1]: import sequences
In [2]: # Import the file sequences.py
In [3]: dir(sequences)
Out [3]: ['__name__', 'fib', 'fib2']
```

Outline

Our Own Module

- Let us create our module "sequences" and save it as "sequences.py"

```
"""A module with different sequences"""

def fib(n):
    """
    prints the Fibonacci sequence from 1 to n
    """
    a, b = 0, 1
    while b < n:
        print b,
        a, b = b, a+b
    print

def fib2(n):
    """
    returns the Fibonacci sequence until n
    """
    res = []
    a, b = 0, 1
    while b < n:
        res.append(b)
        a, b = b, a+b
    return res
```

Our Own Module

- using the module

```
In [1]: import sequences
In [2]: sequences.__name__
Out [2]: 'sequences'
In [3]: sequences.fib(1000)
1 1 2 3 5 8 13 21 34 55 144 233 377 610 987
In [4]: help(sequences)
In [5]: help(sequences.fib)
```

- to automatically reload changes:

```
In [1]: %load_ext autoreload
In [2]: %autoreload 2
```

Our Own Module

- making the module executable
- because you want to test the modul
- because you want to run your module without starting the Python interpreter
- add the following to the end of the sequences.py file:

```
if __name__ == "__main__":
    import sys
    # Usage: python sequences.py <ENTER>
    fib(int(sys.argv[1]))
```

- start the Anaconda prompt, change to the folder in which you saved sequences.py, e.g. if the folder name is “C:\some\path”, type
  `cd ‘‘C:\some\path’’`
- then type
  `python sequences.py 1000`

Excercises

14. Create a program that picks a random number between 1 and 2. The player has to guess the number. The player has 6 attempts to guess. After each guess, tell the player whether his number was too high or too low.
15. “Solve” the Syracuse problem numerically: Take an integer $n \geq 1$, repeat the following operation:
    - if the number is even then divide it by two
    - if the number is odd then multiply it by 3 and add 1

    Does the sequence always reach 1?
    (“Solve” is in quotation marks, because proving this result in general is an unsolved problem so far, see `https://en.wikipedia.org/wiki/Collatz_conjecture`. However, we can calculate the solution of the problem for different numerical examples.)

Excercise 14

- Guessing game guess.py:

```
"""
Picks a random number and lets
the player guess
"""


# imports


# ask six times
    # was the number too high?
    # was the number too low?
    # if neither, exit


# if success ...
# or if failure ...
```

Excercise 14

- Guessing game guess.py

```
"""
Picks a random number and lets
the player guess
"""
import random

secret = random.randint(1,20)
print("I␣thought␣of␣a␣number␣between␣1␣and␣20.")

# Ask the player six times
for number_guess in range(1,7):
    print 'What␣number?'
    guess = int(raw_input())  # for Python 3: input()
    # number was too low
    if guess < secret:
        print 'Your␣guess␣is␣too␣low.'
    # number was too high
    elif guess > secret:
        print 'Your␣guess␣is␣too␣high.'
    else:
        break # otherwise exit

if guess == secret: # if success
    print 'Well␣played.'
else: # or if failure...
    print 'No␣luck!'
```

Exercise 15

- define following function, run with different values of starting_value and number_steps

```
def syracuse(starting_value, number_steps):
    u = starting_value
    print u,
    for n in range(number_steps):
        if u % 2 == 0:
            u = u / 2
        else:
            u = 3 * u + 1
        print u,
        if u == 1:
            print 'Converged␣to␣1␣after␣{}'\
                  '␣steps!'.format(n)
            return
    print 'Did␣not␣converge␣in␣{}␣steps!'\
          .format(number_steps)
```

Outline

Data types in Python

Type	Name	Example	Notes
Integer	`int`	a=10	arbitrarily large; a/4=?
Floats	`float`	b=0.35	precision issues; b+0.1=?
Strings	`string`	c="it is a string"	

- Interesting methods for strings: `c.capitalize()`, `c.split()`, `c.replace(a,b)`...

Data structures

tuple

Simple collection of arbitrary objects. Limited methods.

```
t=(1, 2.5, "data") # t=1, 2.5, "data"
```

Note that indexing starts at zero: `t[0]=1`.

Two methods:

1. Count: `t.count("data")=1`
2. Index: `t.index(2.5)=1`

Data structures

list

A collection of arbitrary objects; many methods available.

```
l=[1, 2.5, "data"]
```

Can convert tuples into lists: `l=list(t)`? Multiple methods:

1. Append (even another list):
   `l.append([4,3])=[1,2.5,"data",[4,3]]`
2. Insert before index:
   `l.insert(1,'insert')=[1,'insert', 2.5,"data",[4,3]]`
3. Remove first occurence:
   `l.remove(2.5)=[1,'insert',"data",[4,3]]`
4. "Slice": `l[1:3]=['insert',"data"]`
5. Sort data: `l.sort()`, or non-mutating version `l2=sorted(l)`

Data structures

- Excercise: Play hangman
- rules
  - The computer chooses a word.
  - In each round the player chooses a letter
  - If the letter is in the word, it appears.
  - If not, then the counter increases and the game appraoches its end.

Solution

```
def play_hangman(word, n):
    guess = len(word)*['_']
    while n>0:
        letters = raw_input('{}.␣guesses␣left:␣'\
                  .format(n))
        letter = letters[0]
        if letter in word:
            for i in range(len(word)):
                if word[i] == letter:
                    guess[i] = letter
            print ''.join(guess)
            if '_' not in guess:
                print 'success!'
                return
        else:
            n -= 1
            print 'wrong.␣{}␣guesses␣left.'\
                  .format(n)
    print 'failure!'
```

Data structures

dict

Dictionaries with key-value stores. Unordered and un-sortable. Maps (generally) strings into strings or numbers.

```
d={'Last': 'Doe', 'First':'John', 'Country':'UK'}
```

Multiple methods:

1. `d.keys()=['Last', 'First', 'Country']`
2. `d.values()=['Doe', 'John', 'England']`
3. Mapping in a dictionary: `d['Last']='Doe'`.
4. Setting an item: `d['Country']=US`

Data structures

set

Mathematical sets: unordered collections of objects, repeated only once.

```
s1=set(['a','b','c','d'])
s2=set(['e','b','c','f'])
```

Multiple methods:

1. s1.union(s2)={'a','b','c','d','e','f'}
2. s1.intersection(s2)={'b','c'}
3. s1.difference(s2)={'a','d'}
4. s1.symmetric_difference(s2)={'a','d','e','f'}

Working with matrices: List arrays

```
v=[0.5, 0.75, 1.0, 1.5, 2.0] # vector
m=[v,v,v] # a 3-by-3 matrix
m=[ [0.5, 0.75, 1.0, 1.5, 2.0]
[0.5, 0.75, 1.0, 1.5, 2.0]
[0.5, 0.75, 1.0, 1.5, 2.0]]
```

1. Easy to select rows or single elements.
   Example: $m[1]$ is second row, $m[1][0]$ first element of the second row.
2. Not easy to select columns! (a "row" is the primary element of the list matrix)
3. Works by reference pointers – changes in v are copied everywhere in m.
   Example: $v[0] = -2$. Try out $m =$?

Numpy arrays

We will import the numerical Python library: `numpy`.

```
import numpy as np
v1=np.array([0.5, 0.75, 1.0, 1.5, 2.0]) #ndarray.
```

Vector methods for `numpy.ndarray`:

1. Sum of elements: `v1.sum()=5.75`.
2. Standard deviation: `v1.std()=0.53`.
3. Cumulative sum:
   `v1.cumsum()=array([0.5, 1.25, 2.25, 3.75, 5.75])`
4. Scalar multiplication, powers, square root...:
   `v1*2 = array([1, 1.5, 2.0, 3.0, 4.0])`
   `v1**2 =array([0.25, 0.5625, 1., 2.25, 4.])`

Matrix operations

```
m1=np.array([v1, v1*2])
m1=np.array([[ 0.5, 0.75, 1., 1.5, 2.],
[ 1., 1.5, 2., 3., 4.]])
```

1. Indexing is *(row, column)*: $m1[0, 2]$ is third element of first row.
2. Column sum: `m1.sum(axis=0)=array([1.5, 2.25, 3., 4.5, 6.])`
   Row sum: `m1.sum(axis=1)=?`
3. Cumulative sum: `v1.cumsum()=array([0.5, 1.25, 2.25, 3.75, 5.75])`
4. Initializing a matrix:
   - `np.zeros((r,c,z), dtype='f', order='C')` or `np.ones((r,c,z), dtype='f', order='C')`.
   - Types (optional): `i` is integer, `f` is float, `b` is boolean....
   - Order (optional): how to store elements in memory 'C' is row-wise, 'F' is column-wise.

Matrix operations (more)

```
m1=np.array([v1, v1*2])
m1=array([[0.5, 0.75, 1., 1.5, 2.],
[1., 1.5, 2., 3., 4.]])
```

1. Flattening:
   `m1.ravel()=array([0.5, 0.75, 1., 1.5, 2., 1., 1.5, 2., 3., 4.])`
2. Matrix size: `m1.shape=(2, 5)`
3. Reshape:
   `m1.reshape(5,-1)=array([[0.5, 0.75], [1., 1.5], [2., 1.], [1.5, 2.], [3., 4.]])`
4. Vertical and horizontal stacking: `vstack` and `hstack`.

Vectorization

Advantages of vectorization:

1. compact code, easy to read and understand.
2. faster execution.

```
v20=np.array([2,3,5])
v21=np.array([0.5,0.6,0.2])
# element-by-element sum
v20+v21=array([2.5, 3.6, 5.2])
# broadcasting the scalar.
2*v20+3=array([7, 9, 13])
```

Outline

Numpy functions

Documentation:

`www.docs.scipy.org/doc/numpy/reference/routines.html`

Name	Description
$np.dot(a, b)$	Dot product of a and b
$np.linalg.det(a)$	Determinant of array a
$np.linalg.solve(a, b)$	Solve linear system $ax = b$
$np.linalg.eig(a)$	Eigenvalues of matrix a
$np.sin(x)$, $np.cos(x)$..	Trigonometric functions
$np.exp(x)$, $np.log(x)$,	
$np.power(x1, x2)$, $np.sqrt(x)$	Arithmetic, exponents, logarithms
$np.median(a)$, $np.mean(a)$	
$np.std(a)$, $np.corrcoef(a, b)$	Summary stats of an array