LECTURE 2: FUNCTIONS & TESTING

Introduction to Scientific Python, CME 193
Jan. 16, 2014
Download code from lectures section of:
`web.stanford.edu/~ermartin/Teaching/CME193-Winter15`
Eileen Martin

Feedback

- Course evaluations only help future classes
- If you want something changed, say so!
  - Talk to me
  - Email
  - Anonymous online survey:

    https://www.surveymonkey.com/s/NSVJDDJ

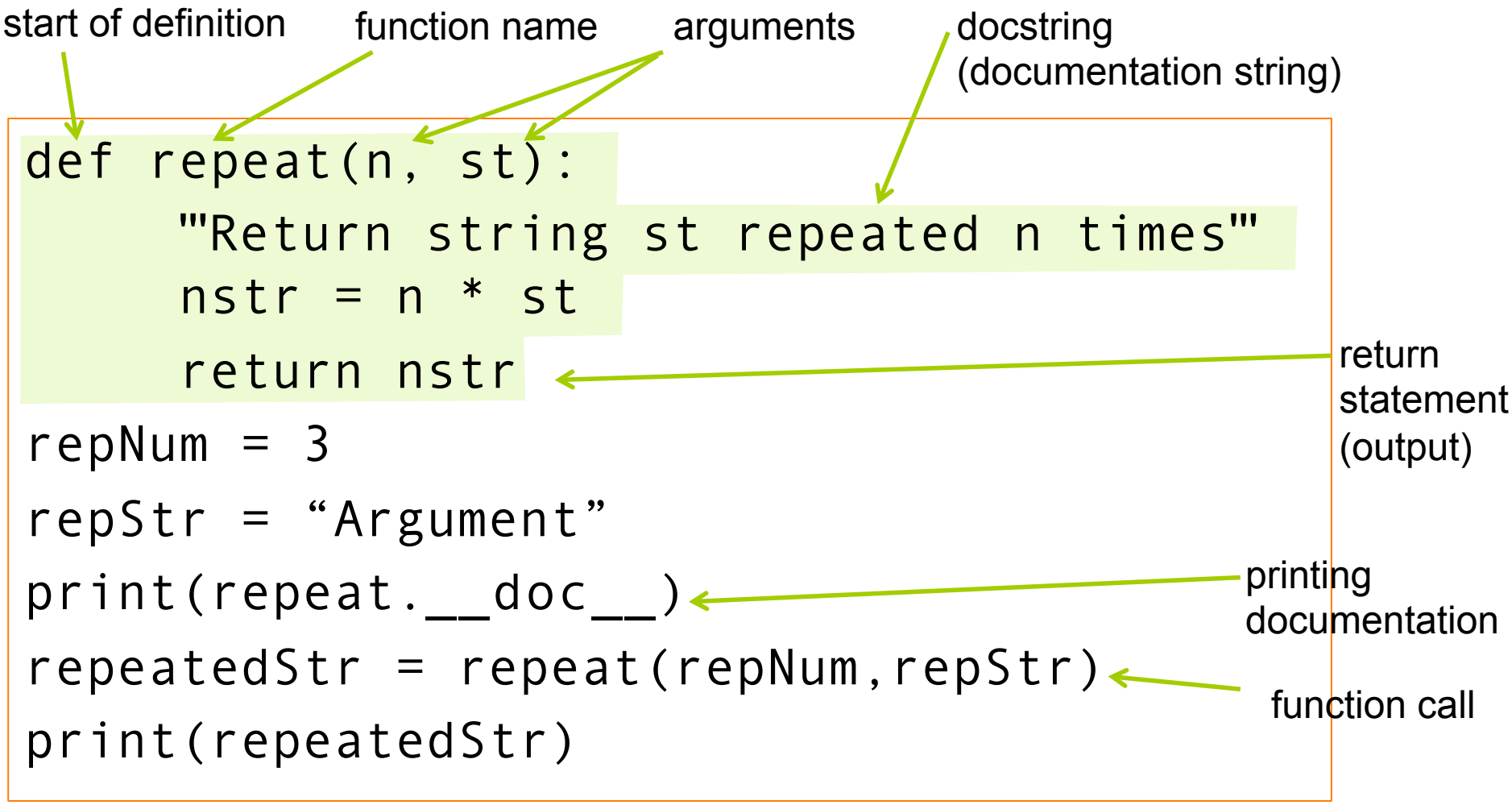
Overview

- Functions
- Scope
- Unit testing
- Q&A on homework 1

What is a function? Why use it?

- **A named set of actions that returns a value**
  - Multiple names could be assigned to the same function
  - May have some arguments/parameters as inputs
    - Arguments are called by some value that is an object reference
  - If there is no return statement, the returned type is NoneType
- A nice way to:
  - **Reuse code** that you want to use multiple times
  - **Organize** your code
- It creates a new local *symbol table* holding a set of variables specific to the function, but can also reference global variables

Anatomy of a function



Organizing functions in separate files

- A function can be called by a script in another file as long as that script knows it can access the functions in that file.
- This is done by importing the function from the module named after that file

rep.py

```
def repeat(n, st):
    '''Return string st
repeated n times'''
    nstr = n * st
    return nstr

def getFirst(st):
    '''Return 1st
character of string st'''
    first = st[0]
    return first
```

runRepeat.py

```
import rep

repNum = 3
repStr = "Argument"
# get description of repeat
print(rep.repeat.__doc__)
# use repeat and print the result
repStr = rep.repeat(repNum,repStr)
print(repStr)
# use getFirst and print the result
print(rep.getFirst(repStr))
```

Different ways to import functions

rep.py

```
def repeat(n, st):
    nstr = n * st
    return nstr

def getFirst(st):
    first = st[0]
    return first
```

Three versions of runRepeat.py

all of rep module

```
import rep
repNum = 3
repStr = "Argument"
repStr = rep.repeat(repNum,repStr)
print(repStr)
print(rep.getFirst(repStr))
```

individual functions in rep.py

```
from rep import repeat, getFirst
repNum = 3
repStr = "Argument"
repStr = repeat(repNum,repStr)
print(repStr)
print(getFirst(repStr))
```

every function in rep.py

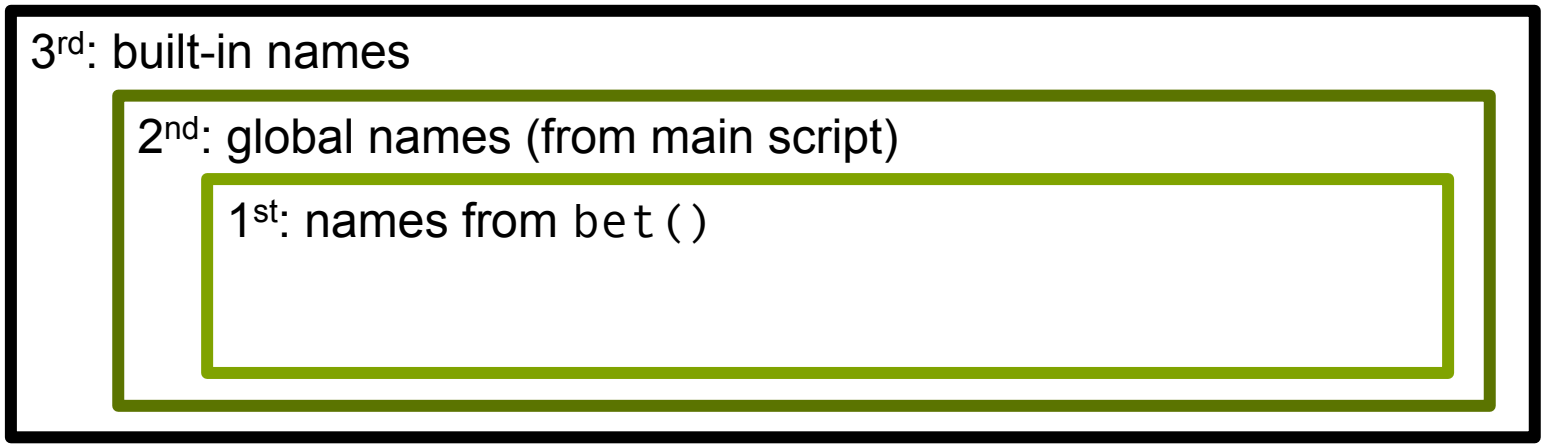
```
from rep import *
repNum = 3
repStr = "Argument"
repStr = repeat(repNum,repStr)
print(repStr)
print(getFirst(repStr))
```

Overview

- Functions
- Scope
- Unit testing
- Q&A on homework 1

Scope: Where do the values come from?

- A function creates a new local *symbol table* holding a set of variables specific to the function, but can also reference global variables.
- How Python finds values of variables when executing some function `bet()`:



Scope

- Variables defined in a function are only accessible in that function.
- Try running scope1.py:
- What is the output of each print statement to the right?

```
def bet(p, winnings, cost):
    "Decide whether to make bet
with some cost and probability
p of some winnings"
    expectation = p * winnings
    choice = False
    if expectation > cost:
        choice = True
    return choice

win = 100
prob = 0.2
choice = True
toBetOrNot = bet(prob,win,25)
print(toBetOrNot)
print(choice)
```

Scope

- Global variables can be referenced from a function
- If a global variable is modified in a function:
  - Added to local symbol table
  - The global value isn't modified
- Modify scope1.py:
- Where does globalVariable value come from in both print statements?

```
def bet(p, winnings, cost):
    "Decide whether to make bet
with some cost and probability
p of some winnings"
    expectation = p * winnings
    print(globalVariable)
    choice = False
    if expectation > cost:
        choice = True
    return choice

win = 100
prob = 0.2
choice = True
globalVariable = True
toBetOrNot = bet(prob,win,25)
print(toBetOrNot)
print(choice)
print(globalVariable)
```

Scope

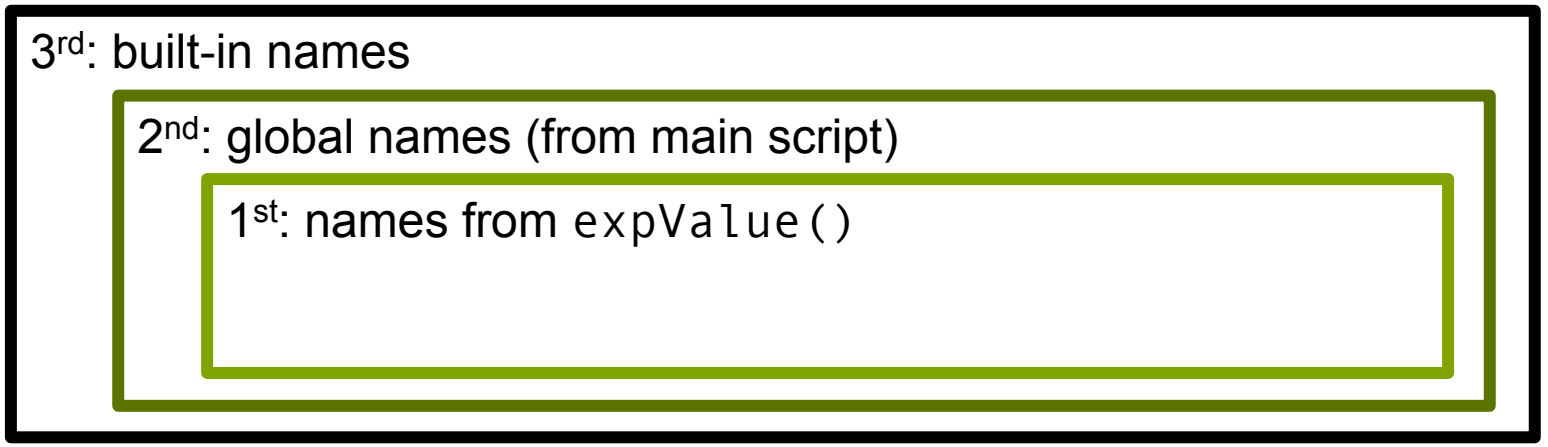
- Global variables can be referenced from a function
- If a global variable is modified in a function:
  - Added to local symbol table
  - The global value isn't modified
- Modify scope1.py:
- Why does this produce an error? Compare to previous two examples.

```
def bet(p, winnings, cost):
    "Decide whether to make bet
with some cost and probability
p of some winnings"
    expectation = p * winnings
    print(choice)
    choice = False
    if expectation > cost:
        choice = True
    return choice

win = 100
prob = 0.2
choice = True
globalVariable = True
toBetOrNot = bet(prob,win,25)
print(toBetOrNot)
print(choice)
print(globalVariable)
```

Scope: when one function calls another and they're defined at same level

- If function `bet()` called `expValue()` defined at the same level, there would be a new symbol table for `expValue()`.

- How Python finds values when executing `expValue()`:



Scope example: 2 functions at same level

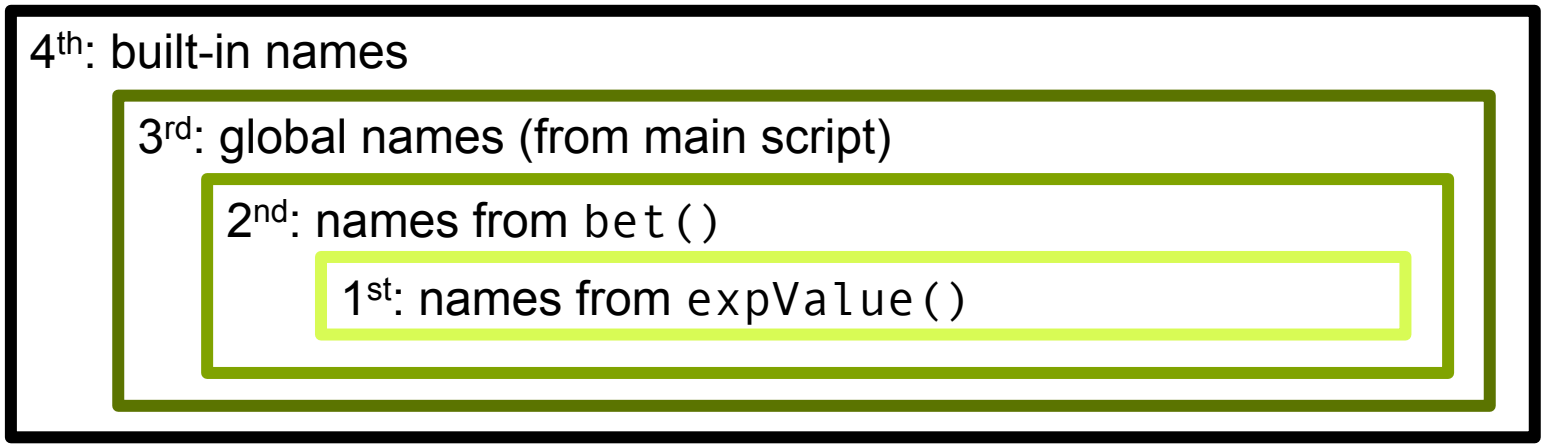
- Try this code in scope2.py:
- What is the output of each print statement?

```
def bet(p, winnings, cost):
    '''Decide whether to make bet with
some cost and probability p of some
winnings'''
    someGlobalVar = 'other test string'
    expectation = expValue(p, winnings)
    choice = False
    if expectation > cost:
        choice = True
    return choice

def expValue(p, winnings):
    '''Expected value of bet with
probability p of some winnings'''
    print(someGlobalVar)
    return p*winnings

win = 100
prob = 0.2
someGlobalVar = 'some test string'
toBetOrNot = bet(prob,win,25)
print(toBetOrNot, someGlobalVar)
```

Scope: when one function is defined inside another

- If function `bet()` called `expValue()` which were defined inside of `bet()`, there would be a new symbol table for `expValue()`, but it could reference values from `bet()`.

- How Python finds values when executing `expValue()`:



Scope example: Function in a function

- Try this code in scope3.py:
- Compare to previous example.
- Note: Can't call `expValue()` from main script

```
def bet(p, winnings, cost):
    '''Decide whether to make bet with
some cost and probability p of some
winnings'''
    def expValue(p, winnings):
        '''Expected value of bet with
probability p of some winnings'''
        print(someGlobalVar)
        return p*winnings
    someGlobalVar = 'other test string'
    expectation = expValue(p, winnings)
    choice = False
    if expectation > cost:
        choice = True
    return choice

win = 100
prob = 0.2
someGlobalVar = 'some test string'
toBetOrNot = bet(prob,win,25)
print(toBetOrNot, someGlobalVar)
```

Overview

- Functions
- Scope
- Unit testing
- Q&A on homework 1

What is unit testing?

- Unit testing means running some checks that certain parts of your code work.
- Unit tests should:
  - each answer one specific question
  - be reproducible/repeatable
  - be easy to run quickly
- Simplest form is to run a bunch of scripts

Example: testing bet() and expValue()

- The function definitions below are in a script called `betting.py`
- How would you propose testing various parts of this code?

```
def bet(p, winnings, cost):
    expectation = expValue(p,winnings)
    choice = False
    if expectation > cost:
        choice = True
    return choice

def expValue(p, winnings):
    return p*winnings
```

Example: testing bet() and expValue()

- One way: run a series of simple scripts
- Here's an example of a couple tests you might want to run (you might want a more thorough testing environment):

```
from betting import *

passed = 0
fail = 0
if expValue(0.1,100) != 10:
   fail += 1
   print('expValue not returning correct expectation')
else:
   passed += 1
if bet(0.1,100,11):
   fail += 1
   print('Decision process may be flawed')
else:
   passed += 1
print('Passed '+str(passed)+' tests of '+str(passed+fail))
```

Basic organization with `unittest`

- As you create a more thorough set of tests, you should have tests organized as functions. This can be done with `TestCase` in the `unittest` module.
- Try this:

```
from betting import *
import unittest

class bettingTests(unittest.TestCase):
   def testPositiveCheck(self):
      '''check exp. value for positive prob. and winnings'''
      self.assertAlmostEqual(expValue(0.1,100),10.0)
   def testNegativeCheck(self):
      '''check exp. value for prob > 0, winnings < 0'''
      self.assertAlmostEqual(expValue(0.1,-1.5),-0.15)
   def SimpleCheck(self):  # doesn't follow name convention
      '''check that you don't bet when cost is too big'''
      self.assertEqual(False, bet(0.1,100,11))

if __name__ == '__main__':
   # run all the unit tests defined in this file
   # with names that start with test
   unittest.main()
```

Further organization with `unittest`

- What if you have multiple tests for the same subset of code, or the same type of potential issue?
- What if you only want some subset of your tests to be run?
- The `unittest` module in the Python Standard Library provides a nice framework for doing this
  - Test cases are individual tests to run
  - Multiple test cases that are similar may be grouped into a class
  - Test suites may contain several test cases
  - Test runners may run multiple test suites

unittest example:

Define a class with two tests

Define a class with one test

Create a suite including 2 of 3 tests above

Create a runner and run and give feedback on `bettingSuite()`

```
from betting import *
import unittest

class expectation(unittest.TestCase):
   def positiveCheck(self):
      '''check exp. value for positive prob. and winnings'''
      self.assertAlmostEqual(expValue(0.1,100),10.0)
   def negativeCheck(self):
      '''check exp. value for prob > 0, winnings < 0'''
      self.assertAlmostEqual(expValue(0.1,-1.5),-0.15)

class decision(unittest.TestCase):
   def simpleCheck(self):
      '''check that you don't bet when cost is too big'''
      self.assertEqual(False, bet(0.1,100,11))

def bettingSuite():
   bettingSuite = unittest.TestSuite()
   bettingSuite.addTest(expectation('positiveCheck'))
   bettingSuite.addTest(decision('simpleCheck'))
   return bettingSuite

if __name__ == '__main__':
   runner = unittest.TextTestRunner()
   runner.run(bettingSuite())
```

Additional features in `unittest`

- You may need to do some set up at the start of many tests
  - Override the setUp() method
- You may also need to dispose of some objects or data
  - Override the tearDown() method

```
from betting import *
import unittest

class expectation(unittest.TestCase):
   def setUp(self):
      self.p = 0.1
   def positiveCheck(self):
      self.assertAlmostEqual(expValue(self.p,100),10.0)
   def negativeCheck(self):
      self.assertAlmostEqual(expValue(self.p,-1.5),-0.15)

if __name__ == '__main__':
   unittest.main()
```

Overview

- Functions
- Scope
- Unit testing
- Q&A on homework 1

First assignment

- Questions about first assignment?
- Solutions will be posted on CourseWork. Please do not share solutions with people who will take the course in the future.
- Suggestions for types of problems you'd like to see in future assignments?
  - Research-inspired problems
  - Problems to explore additional topics