Python R training course - Session 5

Van-Duyet LE, Thinh PHAM

August 27, 2017

Lecture list

1. Introduction to Python
2. The Python Standard Library, if/loops statements
3. Vector/matrix structures, `numpy` library
4. Python data types, File Processing, `Pandas` library
5. **Functions in Python, Debugging.**
   - Lambda function.
   - Recursion.
   - Common errors in Python.
6. Introduction to R, R for Python programmers.
7. Import data, plot data.
8. Data Mining in Python/R.

1 Lambda function

1.1 Syntax of Lambda Function

In Python, anonymous function (lambda function) is a function that is defined without a name.

While normal functions are defined using the **def** keyword, in Python anonymous functions are defined using the **lambda** keyword. Hence, anonymous functions are also called lambda functions.

Syntax:

```
lambda arguments: expression
```

Example:

```
>>> S = lambda a, b: a + b
>>> S(5, 7)
    12
```

```
>>> double = lambda x: x * 2
>>> double(5)
    10
```

Lambda functions can have any number of arguments but only one expression. The expression is evaluated and returned. Lambda functions can be used wherever function objects are required.

In the above program, **lambda x: x * 2** is the `lambda function`. Here **x** is the argument and **x * 2** is the expression that gets evaluated and returned.

This function has no name. It returns a function object which is assigned to the identifier **double**. We can now call it as a normal function. The statement

```
double = lambda x: x * 2
```

is nearly the same as

```
def double(x):
    return x * 2
```

1.2 Use of Lambda Function

We use lambda functions when we require a nameless function for a short period of time.

In Python, we generally use it as an argument to a higher-order function (a function that takes in other functions as arguments). Lambda functions are used along with built-in functions like `filter()`, `map()` etc.

Example use with filter()

The **filter()** function in Python takes in a function and a list as arguments.

The function is called with all the items in the list and a new list is returned which contains items for which the function evaluats to `True`.

Here is an example use of filter() function to filter out only even numbers from a list.

```
>>> def is_odd(x):
>>>     return x % 2 == 0
>>>
>>> my_list = [1, 5, 4, 6, 8, 11, 3, 12]
>>> new_list = list(filter(is_odd, my_list))
    [4, 6, 8, 12]
```

Instead, we can use lambda function:

```
>>> my_list = [1, 5, 4, 6, 8, 11, 3, 12]
>>> new_list = list(filter(lambda x: x % 2 == 0, my_list))
    [4, 6, 8, 12]
```

Example use with map()

The **map()** function in Python takes in a function and a list.

The function is called with all the items in the list and a new list is returned which contains items returned by that function for each item.

Here is an example use of **map()** function to double all the items in a list.

```
>>> my_list = [1, 5, 4, 6, 8, 11, 3, 12]
>>> new_list = list(map(lambda x: x ** 2 , my_list))
    [1, 25, 16, 36, 64, 121, 9, 144]
```

Example use with Pandas DataFrames
For example, we have the dataframe:

```
In [10]: import pandas as pd
         from StringIO import StringIO

         df = pd.read_csv(StringIO("col1,col2,col3\na,b,1\na,b,2\nc,d,3"))
         df

Out[10]:   col1 col2  col3
         0    a    b     1
         1    a    b     2
         2    c    d     3
```

Now, we'll start new **col3_double** from **col3** by apply the double function.

```
In [12]: def double(x):
             return x * 2

         df['col3_double'] = df.col3.apply(double)
         df

Out[12]:   col1 col2  col3  col3_double
         0    a    b     1            2
         1    a    b     2            4
         2    c    d     3            6
```

More quickly with **lambda function** here:

```
In [14]: df['col3_double'] = df.col3.apply(lambda x: x * 2)
         df

Out[14]:   col1 col2  col3  col3_double
         0    a    b     1            2
         1    a    b     2            4
         2    c    d     3            6

In [18]: df.col3.map(lambda t: t > 2)

Out[18]: 0    False
         1    False
         2     True
         Name: col3, dtype: bool
```

2 Python Recursion

Recursion is the process of defining something in terms of itself.

We know that in Python, a function can call other functions. It is even possible for the function to call itself. These type of construct are termed as recursive functions.

Following is an example of recursive function to find the factorial of an integer.

Factorial of a number is the product of all the integers from 1 to that number. For example, the factorial of 6 (denoted as 6!) is `1*2*3*4*5*6 = 720`.

```
In [20]: def calc_factorial(x):
             """This is a recursive function
             to find the factorial of an integer"""

             if x == 1:
                 return 1
             else:
                 return (x * calc_factorial(x-1))

         num = 4
         print "The factorial of", num, "is", calc_factorial(num)
```

```
The factorial of 4 is 24
```

In the above example, **calc_factorial()** is a recursive functions as it calls itself.

When we call this function with a positive integer, it will recursively call itself by decreasing the number.

Each function call multiples the number with the factorial of number 1 until the number is equal to one. This recursive call can be explained in the following steps.

```
calc_factorial(4)              # 1st call with 4
4 * calc_factorial(3)          # 2nd call with 3
4 * 3 * calc_factorial(2)      # 3rd call with 2
4 * 3 * 2 * calc_factorial(1)  # 4th call with 1
4 * 3 * 2 * 1                  # return from 4th call as number=1
4 * 3 * 2                      # return from 3rd call
4 * 6                          # return from 2nd call
24                             # return from 1st call
```

Our recursion ends when the number reduces to 1. This is called the base condition.

Every recursive function must have a base condition that stops the recursion or else the function calls itself infinitely.

- Advantages of recursion
  - Recursive functions make the code look clean and elegant.
  - A complex task can be broken down into simpler sub-problems using recursion.
  - Sequence generation is easier with recursion than using some nested iteration.
- Disadvantages of recursion

- Sometimes the logic behind recursion is hard to follow through.
- Recursive calls are expensive (inefficient) as they take up a lot of memory and time.
- Recursive functions are hard to debug.

Ex1: Python Program to Display Fibonacci Sequence Using Recursion
A Fibonacci sequence is the integer sequence of 0, 1, 1, 2, 3, 5, 8....

```
def fibo_recursion(n):
    # your code goes here

for i in range(10):
    print fibo(i),

# Output: 0 1 1 2 3 5 8 13 21 34
```

Ex2: Find Sum of Natural Numbers Using Recursion

```
def sum_recursion(n):
    # your code goes here

print sum_recursion(10) # Sum of 1 + 2 + 3 + ... + 10

# Output: 55
```

3 Python Errors and Built-in Exceptions

When writing a program, we, more often than not, will encounter errors.

Error caused by not following the proper structure (syntax) of the language is called syntax error or parsing error.

```
>>> if a < 3
  File "<interactive input>", line 1
    if a < 3
           ^
SyntaxError: invalid syntax
```

We can notice here that a colon is missing in the if statement.

Errors can also occur at runtime and these are called exceptions. They occur, for example, when a file we try to open does not exist (`FileNotFoundError`), dividing a number by zero (`ZeroDivisionError`), module we try to import is not found (`ImportError`) etc.

Different kinds of errors can occur in a program, and it is useful to distinguish among them in order to track them down more quickly:

- Syntax errors are produced by Python when it is translating the source code into byte code. They usually indicate that there is something wrong with the syntax of the program. Example: Omitting the colon at the end of a def statement yields the somewhat redundant message SyntaxError: invalid syntax.

- Runtime errors are produced by the runtime system if something goes wrong while the program is running. Most runtime error messages include information about where the error occurred and what functions were executing. Example: An infinite recursion eventually causes a runtime error of "maximum recursion depth exceeded."
- Semantic errors are problems with a program that compiles and runs but doesn't do the right thing. Example: An expression may not be evaluated in the order you expect, yielding an unexpected result.

The first step in debugging is to figure out which kind of error you are dealing with. Although the following sections are organized by error type, some techniques are applicable in more than one situation.

Python Built-in Exceptions:

Exception	Cause of Error
AssertionError	Raised when assert statement fails.
AttributeError	Raised when attribute assignment or reference fails.
EOFError	Raised when the input() functions hits end-of-file condition.
FloatingPointError	Raised when a floating point operation fails.
GeneratorExit	Raise when a generator's close() method is called.
ImportError	Raised when the imported module is not found.
IndexError	Raised when index of a sequence is out of range.
KeyError	Raised when a key is not found in a dictionary.
KeyboardInterrupt	Raised when the user hits interrupt key (Ctrl+c or delete).
MemoryError	Raised when an operation runs out of memory.
NameError	Raised when a variable is not found in local or global scope.
NotImplementedError	Raised by abstract methods.
OSError	Raised when system operation causes system related error.
OverflowError	Raised when result of an arithmetic operation is too large to be represented.
ReferenceError	Raised when a weak reference proxy is used to access a garbage collected referent.
RuntimeError	Raised when an error does not fall under any other category.
StopIteration	Raised by next() function to indicate that there is no further item to be returned by iterator.
SyntaxError	Raised by parser when syntax error is encountered.
IndentationError	Raised when there is incorrect indentation.
TabError	Raised when indentation consists of inconsistent tabs and spaces.
SystemError	Raised when interpreter detects internal error.
SystemExit	Raised by sys.exit() function.
TypeError	Raised when a function or operation is applied to an object of incorrect type.
UnboundLocalError	Raised when a reference is made to a local variable in a function or method, but no value has been bound to that variable.
UnicodeError	Raised when a Unicode-related encoding or decoding error occurs.
UnicodeEncodeError	Raised when a Unicode-related error occurs during encoding.
UnicodeDecodeError	Raised when a Unicode-related error occurs during decoding.
UnicodeTranslateError	Raised when a Unicode-related error occurs during translating.
ValueError	Raised when a function gets argument of correct type but improper value.
ZeroDivisionError	Raised when second operand of division or modulo operation is zero.

3.1 Syntax errors

Syntax errors are usually easy to fix once you figure out what they are. Unfortunately, the error messages are often not helpful. The most common messages are **SyntaxError: invalid syntax** and **SyntaxError: invalid token**, neither of which is very informative.

On the other hand, the message does tell you where in the program the problem occurred. Actually, it tells you where Python noticed a problem, which is not necessarily where the error is. Sometimes the error is prior to the location of the error message, often on the preceding line.

If you are building the program incrementally, you should have a good idea about where the error is. It will be in the last line you added.

If you are copying code from a book, start by comparing your code to the book's code very carefully. Check every character. At the same time, remember that the book might be wrong, so if you see something that looks like a syntax error, it might be.

Here are some ways to avoid the most common syntax errors:

- Make sure you are not using a Python keyword for a variable name.
- Check that you have a colon at the end of the header of every compound statement, including for, while, if, and def statements.
- Check that indentation is consistent. You may indent with either spaces or tabs but it's best not to mix them. Each level should be nested the same amount.
- Make sure that any strings in the code have matching quotation marks.
- If you have multiline strings with triple quotes (single or double), make sure you have terminated the string properly. An unterminated string may cause an invalid token error at the end of your program, or it may treat the following part of the program as a string until it comes to the next string. In the second case, it might not produce an error message at all!
- An unclosed bracket (, {, or [makes Python continue with the next line as part of the current statement. Generally, an error occurs almost immediately in the next line.
- Check for the classic = instead of == inside a conditional.

If nothing works, move on to the next section...

3.2 Runtime errors

Once your program is syntactically correct, Python can import it and at least start running it. What could possibly go wrong?

3.2.1 My program hangs.

If a program stops and seems to be doing nothing, we say it is "hanging." Often that means that it is caught in an infinite loop or an infinite recursion.

- If there is a particular loop that you suspect is the problem, add a print statement immediately before the loop that says "entering the loop" and another immediately after that says "exiting the loop."
- Run the program. If you get the first message and not the second, you've got an infinite loop. Go to the "Infinite Loop" section below.
- Most of the time, an infinite recursion will cause the program to run for a while and then produce a "RuntimeError: Maximum recursion depth exceeded" error. If that happens, go to the "Infinite Recursion" section below.

- If you are not getting this error but you suspect there is a problem with a recursive method or function, you can still use the techniques in the "Infinite Recursion" section.
- If neither of those steps works, start testing other loops and other recursive functions and methods.

3.2.2 Infinite Loop

If you think you have an infinite loop and you think you know what loop is causing the problem, add a `print` statement at the end of the loop that prints the values of the variables in the condition and the value of the condition.

```
while x > 0 and y < 0 :
  # do something to x
  # do something to y

  print  "x: ", x
  print  "y: ", y
  print  "condition: ", (x > 0 and y < 0)
```

Now when you run the program, you will see three lines of output for each time through the loop. The last time through the loop, the condition should be false. If the loop keeps going, you will be able to see the values of x and y, and you might figure out why they are not being updated correctly.

3.2.3 Infinite Recursion

Most of the time, an infinite recursion will cause the program to run for a while and then produce a Maximum recursion depth exceeded error.

3.2.4 Flow of Execution

If you are not sure how the flow of execution is moving through your program, add print statements to the beginning of each function with a message like "entering function foo," where foo is the name of the function.

Now when you run the program, it will print a trace of each function as it is invoked.

4 References

- http://www.greenteapress.com/thinkpython/thinkCSpy/html/app01.html
- http://www.python-course.eu/python3_lambda.php