Python R training course - Session 1

Van-Duyet LE, Thinh PHAM

August 17, 2017

Python is nowadays become an active programming language used for scientic computing, especially it is widely used in data science. Moreover, Pythonis totally free and prefered by a lot of companies. Based on this signicantneed, we aim to provide the JVN students the most important knowledgein Python.

Adapted by Volodymyr Kuleshov and Isaac Caswell from the CS231n

Python tutorial by Justin Johnson (`http://cs231n.github.io/python-numpy-tutorial/`). The theory is from MIT Opencourseware: A Gentle Introduction to Programming Using Python (http://ocw.mit.edu)

Lecture list

1. **Introduction to Python**
   - What and Why is Python: the signicance of Python in computer science, data science community and other fields.
   - How to setup and access Python environment (e.g: install, start, stop). Students are required to be familiar with using command line to carry out some code.
   - Simple Python convention (variables, constants, operations) and coding convention (naming variables).
   - Function
2. The Python Standard Library, if/loops statements
   - Some useful and basic built-in functions (abs, etc.)
   - How to use import to import libraries (ex: math, random). We will introduce you several basic libraries that are often used in Python.
   - Introduce Comparison and Boolean Operators (if/else/while/for)
   - Students are also introduced the concept of object-orientation in Python.
3. Vector/matrix structures, `numpy` library
   - Vector/matrix: structures in Python, some built-in functions, Operators for vector/matrix (access, comparation, search)
   - Introduce to module Numpy
4. Python data types, File Processing, `Pandas` library
   - Introduce String/List and basic built-in functions and operators for String/List.
   - Structure of Python files. Read/create a file.

- Read csv file using `Pandas` library

5. Functions in Python, Debugging.
   - What is functions and how to define functions in Python
   - Passing Arguments to Functions
   - Common errors in Python
6. Introduction to R, R for Python programmers.
   - From Python to R.
   - Variable, using library, function in R
7. Import data, plot data.
   - Read csv file, quick summary.
   - Plot data.
8. Data Mining in Python/R.

1 Introduction to Python

Python is a great general-purpose programming language on its own, butwith the help of a few popular libraries (*numpy, scipy, matplotlib*) it becomesa powerful environment for scientific computing.

1.1 Why we choose Python

1.1.1 Popular

Python is a general-use high-level programming language that bills itself aspowerful, fast, friendly, open, and easy to learn. Python "plays well withothers" and "runs everywhere".

Bank of America uses Python to crunch financial data. The TheoreticalPhysics Division of Los Alamos National Laboratory chose Python to notonly control simulations, but also analyze and visualize data. Facebookturns to the Python library Pandas for its data analysis because it sees thebenefit of using one programming language across multiple applications.

1.2 Simple

Python is easy to use, powerful, and versatile, making it a great choice for beginners and experts alike. Python's readability makes it a great first programming language — it allows you to think like a programmer and not waste time understanding the mysterious syntax that other programming languages can require.

1.3 Write once run every where

Supporting Operate System : Windows, Linux/Unix, MacOS

1.4 References

- http://www.mastersindatascience.org/data-scientist-skills/python/
- https://www.codeschool.com/blog/2016/01/27/why-python/
- https://www.quora.com/Why-is-Python-a-language-of-choice-for-data-scientists
- http://www.kdnuggets.com/2015/05/r-vs-python-data-science.html

2 Install Python

We highly recommend all of you install Anaconda Python `https://www.continuum.io/downloads`

ANACONDA
Powered by Continuum Analytics
Anaconda Cloud
Documentation
Blog
Contact
DOWNLOAD
What Is Anaconda?
Products
Support & Solutions
Community
About
Resources
DOWNLOAD ANACONDA DISTRIBUTION
Version: 4.4.0 | Release Date: May 31, 2017
Download for:
High-Performance Distribution
Easily install 1,000+ data science packages
Package Management
Manage packages, dependencies and environments with conda
Portal to Data Science
Uncover insights in your data and create interactive visualizations

Benefit when using Python Anaconda:

- Easy setup and install a stable environment for Python programming.
- 720+ data science packages for interactive data visualizations, machine learning, deep learning and Big Data.
- World-class package, dependency and environment management.

3 Basics of Python

Python is a high-level, dynamically typed multiparadigm programming language. Python code is often said to be almost like pseudocode, since it allows you to express very powerful ideas in very few lines of code while being very readable. As an example, here is an implementation of the classic quicksort algorithm in Python

```
In [1]: def quicksort(arr):
            '''
```

```
        quicksort function
        '''
        if len(arr) <= 1:
            return arr

        pivot = arr[len(arr) / 2]
        left = [x for x in arr if x < pivot]
        middle = [x for x in arr if x == pivot]
        right = [x for x in arr if x > pivot]

        return quicksort(left) + middle + quicksort(right)

    print quicksort([3,6,8,10,1,2,1])
```

```
[1, 1, 2, 3, 6, 8, 10]
```

There are currently two different supported versions of Python, 2.7 and 3.4. Somewhat confusingly, Python 3.0 introduced many backwards-incompatible changes to the language, so code written for 2.7 may not work under 3.4 and vice versa. For this class all code will use Python 2.7.

You can check your Python version at the command line by running `python --version`.

3.1 Variables, expressions and statements

Reference: http://www.greenteapress.com/thinkpython/thinkCSpy/html/chap02.html

3.1.1 Values and types

A value is one of the fundamental things like a letter or a number that a program manipulates. The values we have seen so far are 2 (the result when we added `1 + 1`), and `'Hello, World!'`.

```
In [2]: 1 + 3

Out[2]: 4

In [3]: 'Hello, World!'

Out[3]: 'Hello, World!'
```

These values belong to different **types**: 2 is an **integer**, and `'Hello, World!'` is a **string**, so-called because it contains a "string" of letters.

The print statement also works for integers:

```
In [4]: print 2
```

```
2
```

If you are not sure what type a value has, the interpreter can tell you.

```
In [5]: print type('17')
        print type(17)

<type 'str'>
<type 'int'>
```

3.1.2 Variables

A variable is a name that refers to a value. The **assignment statement** creates new variables and gives them values:

```
In [6]: message = "JVN"
        n = 10

In [7]: print message

JVN


In [8]: print "Hi, " + message

Hi, JVN
```

Variable names and keywords

Programmers generally choose names for their variables that are meaningful they document what the variable is used for.

Variable names can be arbitrarily long. They can contain both letters and numbers, but:

- they have to begin with a letter. Although it is legal to use uppercase letters, by convention we don't.
- `JVN` and `Jvn` are different variables.
- the underscore character (_) can appear in a name. Such as `my_name` or `_n`

If you give a variable an illegal name, you get a syntax error:

```
In [9]: 76trombones = 'big parade'


          File "<ipython-input-9-a5f509298d77>", line 1
        76trombones = 'big parade'
                  ^
    SyntaxError: invalid syntax



In [10]: while = 1000000
```

```
      File "<ipython-input-10-07be154a2b96>", line 1
    while = 1000000
          ^
SyntaxError: invalid syntax
```

It turns out that class is one of the Python **keywords**. Keywords define the language's rules and structure, and they cannot be used as variable names.

Python has twenty-nine keywords:

```
and       def      exec     if       not      return
assert    del      finally  import   or       try
break     elif     for      in       pass     while
class     else     from     is       print    yield
continue  except   global   lambda   raise
```

3.1.3 Statements

A statement is an instruction that the Python interpreter can execute. We have seen two kinds of statements: print and assignment.

When you type a statement on the command line, Python executes it and displays the result, if there is one. The result of a print statement is a value. Assignment statements don't produce a result.

A script usually contains a sequence of statements. If there is more than one statement, the results appear one at a time as the statements execute.

For example, the script

```
>>> print 1
>>> x = 2
>>> print x
```

produces the output

```
1
2
```

Again, the assignment statement produces no output.

3.1.4 Operators and operands

Operators are special symbols that represent computations like addition and multiplication. The values the operator uses are called operands.

The following are all legal Python expressions whose meaning is more or less clear:

```
20+32   hour-1   hour*60+minute   minute/60   5**2   (5+9)*(15-7)
```

Operators:

- Arithmatic: +, -, *, /, and % (modulus)
- Comparison: ==, !=, <, >, <=, >=
- Logical: and, or, not
- Exponentiation: **

3.2 Functions

3.2.1 Function calls

You have already seen one example of a function call:

```
>>> type("32")
<type 'str'>
```

- **type** is function name, it displays the type of a value or variable.
- **"32"** is **argument** of the function
- `<type 'str'>` The result is called the return value.

Instead of printing the return value, we could assign it to a variable:

```
>>> betty = type("123456")
>>> print betty
<type 'str'>
```

Some build-in function of python: https://docs.python.org/2/library/functions.html

```
abs()       divmod()    input()         open()      staticmethod()
all()       enumerate() int()           ord()       str()
any()       eval()      isinstance()    pow()       sum()
bin()       file()      iter()          property()  tuple()
bool()      filter()    len()           range()     type()
dir()       id()        oct()           sorted()
...
```

```
In [11]: print abs(-10)

10
```

```
In [12]: print len("JVN Forever")

11
```

3.2.2 Type conversion & Type coercion

Python provides a collection of built-in functions that convert values from one type to another. The int function takes any value and converts it to an integer, if possible, or complains otherwise:

```
In [13]: print int("32")
         print int(3.99999)
```

```
32
3
```

```
In [14]: print int("Hello")
```

```
        ---------------------------------------------------------------------------

        ValueError                                Traceback (most recent call last)

        <ipython-input-14-f14c94228214> in <module>()
    ----> 1 print int("Hello")


        ValueError: invalid literal for int() with base 10: 'Hello'
```

```
In [ ]: # the str function converts to type string:
        str(32)
```

Type coercion:

```
In [15]: minute = 59
         minute / 60
```

```
Out[15]: 0
```

```
In [16]: minute = 59
         float(minute) / 60
```

```
Out[16]: 0.9833333333333333
```

3.2.3 Math functions

Python has a math module that provides most of the familiar mathematical functions. A **module** is a file that contains a collection of related functions grouped together.

Before we can use the functions from a module, we have to import them:

```
>>> import math
```

```
In [17]: import math

         print math.sin(1.5)
```

```
0.997494986604
```

```
In [18]: print math.sqrt(2) / 2.0
```

```
0.707106781187
```

3.2.4 Adding new functions

Creating new functions to solve your particular problems is one of the most useful things about a general-purpose programming language.

```
In [19]: def foo(x):
             x = x + 1
             return x

         foo(10)

Out[19]: 11
```

Python functions are defined using the `def` keyword. For example:

```
In [20]: def sign(x):
             if x > 0:
                 return 'positive'
             elif x < 0:
                 return 'negative'
             else:
                 return 'zero'

         print sign(-1)
         print sign(0)
         print sign(10)

negative
zero
positive
```

```
In [21]: def add_2(number):
             return number + 2

         add_2(10)

Out[21]: 12
```

We will often define functions to take optional keyword arguments, like this:

```
In [22]: def hello(name, loud=False):
             if loud:
                 print 'HELLO, %s' % name.upper()
             else:
                 print 'Hello, %s!' % name

         hello('Bob')
         hello('Fred', loud=True)
```

```
Hello, Bob!
HELLO, FRED
```

Read python document [how to use build-in function with the argument]: https://docs.python.org/2/library/functions.html#len

4 References

- Code convention: PEP 8 — the Style Guide for Python Code - http://pep8.org
- Online Course, code Python in your browser - https://www.datacamp.com
- Python 2.7.13 documentation - https://docs.python.org/2/
- Basic Python Exercises - Google for Education
- [Book] Learning with Python - How to Think Like a Computer Scientist
- Python Numpy Tutorial - http://cs231n.github.io/python-numpy-tutorial/

5 Homework exercise

- Homework reading: Python Classes (https://docs.python.org/2/tutorial/classes.html), list, set
- Try to use 10 build-in functions.

5.1 Exercise

5.1.1 a. Odd Or Even

Ask the user for a number. Depending on whether the number is even or odd, print out an appropriate message to the user.

```
>>> n = input("Input n = ")
>>> odd_even(n)
```

Extras:

- If the number is a multiple of 4, print out a different message.

  ```
  >>> odd_even(7)
    'odd'
  >>> odd_even(4)
    'the number is a multiple of 4'
  ```

- Ask the user for two numbers: one number to check (call it num) and one number to divide by (check). If check divides evenly into num, tell that to the user. If not, print a different appropriate message.

  ```
  >>> check(10,2)
      '10 divides evenly by 2'

  >>> check(11,2)
      '11 does not divide evenly by 2'
  ```

```
In [1]: # Solution
        def odd_even(n):
            if n % 2 == 0:
                return 'odd'
            return 'even'

        # Extra 1
        def odd_even(n):
            if n % 4:
                return 'the number is a multiple of 4'
            if n % 2 == 0:
                return 'odd'
            return 'even'

        # Extra 2
        def check(a, b):
            if a % b == 0:
                return a, 'divides evenly by', b
            return a, 'does not divide evenly by', b
```

5.1.2 b. Max Of Three

Implement a function that takes as input three variables, and returns the largest of the three. Do this function just in 1 or 2 line.

```
>>> max_of_three(5,20,2)
20
```

```
In [3]: # Solution
        def max_of_three(a, b, c):
            return max(max(a, b), c)

        max_of_three(1, 2, 3)

Out[3]: 3
```

5.1.3 c. Max Of Three (without `max()` function)

Implement a function that takes as input three variables, and returns the largest of the three. Do this without using the Python `max()` function!

```
>>> max_of_three(5,20,2)
20
```

```
In [4]: # Solution
        def max_of_three(a, b, c):
            m = a if a > b else b
            m = m if m > c else c
            return m
        max_of_three(1,2,3)

Out[4]: 3
```

5.1.4 d. Calculating Area of Circle

Using `math`, implement a function that calculating the area of circle, given by `radius` arg.

```
>>> calculate_area(10)
    314.159265359
```

```
In [5]: # Solution
        import math

        def calculate_area(r):
            return math.pi * (r**2)

        calculate_area(10)

Out[5]: 314.1592653589793
```

5.1.5 e. Reverse Word Order Solutions

Write a program that asks the user for a long string containing multiple words. Print back to the user the same string, except with the words in backwards order.

```
>>> reverse_str("My name is Duyet")
    "Duyet is name My"
```

```
In [6]: # Solution
        def reverse_str(s):
            s = s.split()
            return ' '.join(reversed(s))

        reverse_str("Le Van Duyet")

Out[6]: 'Duyet Van Le'
```

5.1.6 f. not_string

Given a string, return a new string where **"not"** has been added to the front. However, if the string already begins with "not", return the string **unchanged**.

```
>>> not_string('candy')
    'not candy'

>>> not_string('x')
    'not x'

>>> not_string('not bad')
    'not bad'
```

```
In [8]: # Solution
        def not_string(s):
            s = s.strip()
            if s[:3] == 'not':
                return s
            return 'not ' + s

        not_string("not bad")

Out[8]: 'not bad'
```

```
In [10]: # Solution 2
         def not_string(s):
             s = s.strip()
             if s.find('not') == 0:
                 return s
             return 'not ' + s

         not_string("bad")

Out[10]: 'not bad'
```

5.1.7 g. negative/positive.

Given 2 int values, return **True** if one is **negative** and one is **positive**.

```
>>> pos_neg(-1, 1)
    True

>>> pos_neg(100, 200)
    False
```

```
In [11]: # Solution
         def pos_neg(a, b):
             if a * b < 0:
                 return True
             return False

         pos_neg(-1, 1)

Out[11]: True
```

5.1.8 h. Ahihi

Given a string and a non-negative int **n**, return a larger string that is n copies of the original string.

```
>>> string_times('Hi', 2)
    'HiHi'
```

```
>>> string_times('Hi', 3)
    'HiHiHi'

>>> string_times('Hi', 1)
    'Hi'
```

```
In [12]: # Solution

         def string_times(s, n):
             return s * n

         string_times("Hi", 3)

Out[12]: 'HiHiHi'
```

5.1.9 i. First-half

Given a string of even length, return the first half. So the string **"WooHoo"** yields **"Woo"**.

```
>>> first_half('WooHoo')
    'Woo'

>>> first_half('HelloThere')
    'Hello'

>>> first_half('abcdef')
    'abc'
```

```
In [13]: # Solution
         def first_half(s):
             return s[:len(s) / 2]

         first_half('HelloThere')

Out[13]: 'Hello'
```

5.1.10 j. Number 6

The number 6 is a truly great number. Given two int values, **a** and **b**, return **True** if either one is 6. Or if **their sum** or **difference** is **6**.

Note: the function `abs(num)` computes the absolute value of a number.

```
>>> love6(6, 7)
    True
>>> love6(1, 5)  # Sum = 6
    True
>>> love6(13, 7)  # 13 - 7 = 6
    True
>>> love6(2,9)
    False
```

```
In [17]: # Solution

         def love6(a, b):
             if a == 6 or b == 6:
                 return True
             if abs(a - b) == 6 or (a + b) == 6:
                 return True
             return False

         print love6(6, 7)
         print love6(1, 5)
         print love6(13, 7)
         print love6(2,9)
```

```
True
True
True
False
```