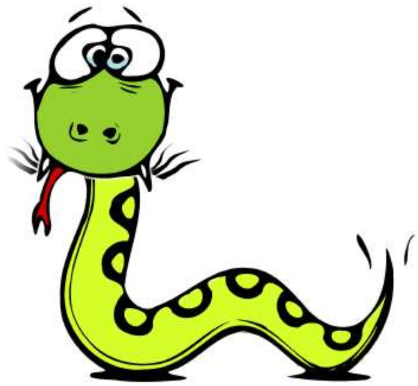
Introduction to Python



Python is a general-purpose programming language which is another way to say that it can be used for nearly everything. In technical terms, Python is an object-oriented, high-level programming language with integrated dynamic semantics primarily for web and app development. Recent developments have extended Python's range of applicability to econometrics, statistics and general numerical analysis. Python – with the right set of add-ons – is comparable to domain-specific languages such as R and MATLAB.

If you want to apply statistical methods, the statistics library of R is second to none, and R is clearly at the forefront in new statistical algorithm development – meaning you are most likely to find that new(ish) procedure in R. Python has less statistics packages, but its performance is better.

Why you should consider Python?

- You need a language which can act as an end-to-end solution so that everything from accessing web-based services and database servers, data management and processing and statistical computation can be accomplished in a single language. Python can even be used to write server-side apps such as dynamic website (see e.g. http://stackoverflow.com), apps for desktop-class operating systems with graphical user interfaces and even tablets and phones apps (iOS and Android).
- Data handling and manipulation – especially cleaning and reformatting – is an important concern. Python is substantially more capable at data set construction than either R or MATLAB.
- Performance is a concern, but not at the top of the list.
- Free is an important consideration.
- Knowledge of Python, as a general-purpose language, is complementary to R or MATLAB.

Scientific/statistical use of python has grown over the years (http://www.burtchworks.com/2017/06/19/2017-sas-r-python-flash-survey-results/), and for some time there have been packages such as scipy, numpy, and matplotlib that help scientific scripting, analysis and visualisation. Python is widely used, including by a number of big companies like Google, Pinterest, Instagram, Disney, Yahoo!, Nokia, IBM, and many others.

Installation of Python: https://www.python.org

- Windows: install with default settings plus Add Python.exe to PATH
    - The default settings come with the IDLE (Integrated Development and Learning Environment) shell which is a default editor that accompanies Python. Coding with it is interactive which is good for beginners.
    - Try our first line of code print("Hello, world!") in IDLE
    - type exit() or Ctrl+C if there is any Syntax Error
- Mac: https://youtu.be/jHV8e9DO5bg or https://docs.python.org/3/using/mac.html#getting-and-installing-macpython

Installation of a Python IDE: for example, https://jupyter.org/

```
pip install jupyterlab
```

```
jupyter-lab
```

Installation of Python packages: https://docs.python.org/3/installing/index.html

For example, to install the Python scientific library Numpy

```
python -m pip install numpy
```

Check the installed libraries:

```
pip list
```

Upgrade a library, for example to upgrade numpy:

```
python -m pip install --upgrade numpy
```

You can write your Python code line by line or compose a *.py file and run the file.

```
Python *.py
```

Try your first Python code:

```
print("Hello, world!")
```

Python Resources:

- Main website http://www.python.org and SciPy site http://scipy.org.
- Official Python Tutorial https://docs.python.org/3/tutorial/index.html .
- Google's Python Class (2 day class materials including video and exercises) https://developers.google.com/edu/python.
- Think Stats - Exploratory Data Analysis in Python (http://greenteapress.com/thinkstats2/thinkstats2.pdf)
- Python scientific lecture notes (http://www.scipy-lectures.org/)
- Learn Python the Hard Way (https://learnpythonthehardway.org/)

Python packages: Python Package Index (PyPI) is the repository of software for Python at http://pypi.python.org/pypi. Once a package is successfully installed, then you can `import` the module within your script. For example,

```
import numpy as np

import matplotlib.pyplot as plt  #import the pyplot module from the package matplotlib

import scipy as sp

import pandas as pd

from numpy import array, sqrt  #import a specific function from a package
```

Use Python as a calculator:

```
5+5

Out[1]: 10

10.5-2*3

Out[2]: 4.5


10**2                    # ** is used to calculate powers

Out[3]: 100


17 % 3                   # the % operator returns the remainder after division

Out[4]: 2
```

Python Data types (http://developer.rhino3d.com/guides/rhinopython/python-datatypes/):

(1) **Numbers:**

Python numbers variables are created by the standard Python method:

```
var = 382
```

Most of the time using the standard Python number type is fine. Python will automatically convert a number from one type to another if it needs. But, under certain circumstances that a specific number type is needed (ie. complex, hexidecimal), the format can be forced into a format by using additional syntax in the table below:

Type	Format	Description
int	a = 10	Signed Integer
float	a = 45.67	(.) Floating point real values
complex	a = 3.14J	(J) complex number.

Most of the time Python will do variable conversion automatically. You can also use Python conversion functions (int(), float(), complex()) to convert data from one type to another. In addition, the function **type()** returns information about how your data is stored within a variable.

```
message = "Good morning"
num = 85
pi = 3.14159

print(type(message))  # This will return a string
print(type(num))  # This will return an integer
print(type(pi))  # This will return a float
```

(2) **String**

Create string variables by enclosing characters in quotes. Python uses single quotes 'double quotes " and triple quotes """ to denote literal strings.

Only the triple quoted strings """ also will automatically continue across the end of line statement.

```
firstName = 'john'

lastName = "smith"

message = """This is a string that will span across multiple lines. Using newline characters

and no spaces for the next lines. The end of lines within this string also count as a newline when printed"""
```

Strings can be accessed as a whole string, or a substring of the complete variable using brackets '[]'. Here are a couple examples:

```
var1 = 'Hello World!'

var2 = 'RhinoPython'


print(var1[0]) # this will print the first character in the string an `H`

print(var2[1:5]) # this will print the substring 'hinoP`
```

(3) **List**

Lists are a very useful variable type in Python. A list can contain an ordered set of values. List variables are declared by using brackets [ ] following the variable name.

```
A = [ ] # This is a blank list variable

B = [1, 23, 45, 67] # this list creates an initial list of 4 numbers.

C = [2, 4, 'john']  # lists can contain different variable types.
```

All lists in Python are zero-based indexed. You can access individual list elements. When referencing a member or the length of a list the number of list elements is always the number shown plus one.

```
mylist = ['Rhino', 'Grasshopper', 'Flamingo', 'Bongo']

B = len(mylist) # This will return the length of the list which is 3. The index is 0, 1, 2, 3.

Print(mylist[1]) # This will return the value at index 1, which is 'Grasshopper'

Print(mylist[0:2]) # This will return the first 3 elements in the list.
```

You can assign data to a specific element of the list using an index into the list. The list index starts at zero. Data can be assigned to the elements of an array as follows:

```
mylist = [0, 1, 2, 3]
mylist[0] = 'Rhino'
mylist[1] = 'Grasshopper'
mylist[2] = 'Flamingo'
mylist[3] = 'Bongo'
print(mylist)
```

You can change an individual list element:

```
mylist[0]= 'MATH'
mylist[1] = 311
print(mylist)
```

Remark: Don't change an element in a string!

```
mylist[:]= []  # this clears the list
print(mylist)
```

Sorting Lists: There are two ways to sort lists:

(a) Use the sorted() built function

```
mylist=[3,31,123,1,5]
print(mylist)
sorted(mylist)
print(mylist)
```

(b) Use the sort() method of lists

```
mylist.sort()
mylist       # The sort() method modifies the list!!!
```

Note. Lists aren't limited to a single dimension. Although most people can't comprehend more than three or four dimensions. You can declare multiple dimensions by separating them with commas. In the following example, the MyTable variable is a two-dimensional array :

```
MyTable = [[], []]
```

In a two-dimensional array, the first number is always the number of rows; the second number is the number of columns.

(4) **Tuple**

Tuples are a group of values like a list and are manipulated in similar ways. But, tuples are fixed in size once they are assigned. In Python the fixed size is considered immutable as compared to a list that is dynamic and mutable. Tuples are defined by parenthesis ().

```
myGroup = ('Rhino', 'Grasshopper', 'Flamingo', 'Bongo')
```

Here are some advantages of tuples over lists:

Elements to a tuple. Tuples have no append or extend method.

Elements cannot be removed from a tuple.

You can find elements in a tuple, since this doesn't change the tuple.

You can also use the in operator to check if an element exists in the tuple.

Tuples are faster than lists. If you're defining a constant set of values and all you're ever going to do with it is iterate through it, use a tuple instead of a list.

It makes your code safer if you "write-protect" data that does not need to be changed.

It seems tuples are very restrictive, so why are they useful? There are many datastructures in Rhino that require a fixed set of values. For instance a Rhino point is a list of 3 numbers [34.5, 45.7, 0]. If this is set as tuple, then you can be assured the original 3 number structure stays as a point (34.5, 45.7, 0). There are other data structures such as lines, vectors, domains and other data in Rhino that also require a certain set of values that do not change. Tuples are great for this.

(5) **Sets**

A set is an unordered collection with no duplicate elements. Set objects support mathematical operations like union, intersection, and difference.

```
a = set('abracadabra')

a

Out[13]: {'d', 'b', 'a', 'r', 'c'}


b = set('alacazam')

b

Out[15]: {'l', 'm', 'z', 'a', 'c'}
```

```
a - b                              # letters in a but not in b
Out[16]: {'b', 'd', 'r'}

a | b                              # letters in a or b or both
Out[17]: {'a', 'b', 'c', 'd', 'l', 'm', 'r', 'z'}

a & b                              # letters in both a and b
Out[18]: {'a', 'c'}

a ^ b                              # letters in a or b but not both
Out[19]: {'l', 'm', 'z', 'd', 'b', 'r'}
```

(6) **Dictionary**

A dictionary is an unordered set of key and value pairs, with the requirement that the keys are unique (within one dictionary). A pair of braces creates an empty dictionary: {}. Placing a comma-separated list of key:value pairs within the braces adds initial key:value pairs to the dictionary; this is also the way dictionaries are written on output.

```
tel = {'jack': 4098, 'sape': 4139}
tel['guido'] = 4127        #Add a new key:value pair to the dictionary
tel
Out[23]: {'jack': 4098, 'sape': 4139, 'guido': 4127}

list(tel.keys())             #list the keys
Out[24]: ['jack', 'sape', 'guido']

tel['sape'] = 4159        #modify a key:value pair
tel
Out[26]: {'jack': 4098, 'sape': 4159, 'guido': 4127}

del tel['sape']   #delete a key from the dictionary
tel
```

```
Out[28]: {'jack': 4098, 'guido': 4127}


sorted(tel.keys())    #sort the keys

Out[29]: ['guido', 'jack']
```

(7) **Array**.

Defined in **numpy**. Vectors and matrices for numerical data manipulation. NumPy (http://www.scipy-lectures.org/) is an extension to the Python programming language, adding support for large, multi-dimensional (numerical) arrays and matrices, along with a large library of high-level mathematical functions to operate on these arrays.

```
import numpy as np
data1 = [1, 2, 3, 4, 5] # list
arr1 = np.array(data1) # 1d array
arr1
Out[38]: array([1, 2, 3, 4, 5])

data2 = [range(1, 5), range(5, 9)] # list of lists
arr2 = np.array(data2) # 2d array

arr2
Out[40]:
array([[1, 2, 3, 4],
       [5, 6, 7, 8]])

arr2[1,1]
Out[41]: 6

arr2[0,0]
Out[42]: 1

arr2[0,1]
Out[43]: 2
```

```
arr2[1,0]
Out[44]: 5

arr2[0, :] # row 0: returns 1d array ([1, 2, 3, 4])
Out[45]: array([1, 2, 3, 4])

arr2[:, 0] # column 0: returns 1d array ([1, 5])
Out[47]: array([1, 5])

arr2[:, :2] # columns strictly before index 2 (2 first columns)
Out[49]:
array([[1, 2],
       [5, 6]])

arr2[:, 2:] # columns after index 2 included
Out[52]:
array([[3, 4],
       [7, 8]])

arr2[:, 1:4] # columns between index 1 (included) and 4 (excluded)
Out[54]:
array([[2, 3, 4],
       [6, 7, 8]])
```

Vectorized operations:

```
nums= np.random.randn(4, 2) # random normals in 4x2 array


nums
Out[59]:
array([[-0.4846231 , -0.80833393],
       [ 0.20384442, -0.01385098],
       [ 0.80863404, -1.34495194],
       [-1.53458325, -0.587191  ]])
nums.mean()
nums.std()
nums.argmin() # index of minimum element
nums.sum()
nums.sum(axis=0)      # sum of rows
nums.sum(axis=1)      # sum of columns
```

Descriptive Statistics

Importing and Exporting Data

Pandas is an increasingly important component of the Python scientific stack; All of the data readers in pandas load data into a pandas DataFrame. The DataFrame is very useful since it includes useful information such as column names read from the data source.

(1) Importing text files

Suppose we have a **Comma-separated value (CSV)** Text file (grades.csv) which looks like

Name	Exam1	Exam2	Exam3	Letter
john	23	46	35	F
mary	42	31	36	F
sam	58	22	55	F
oksana	81	88	79	P
tom	11	19	18	F
peter	55	64	69	P
larisa	81	78	52	P

```
from pandas import read_csv
grades1=read_csv('grades.csv') #add path if necessary
grades1
Out[24]:
     Name  Exam1  Exam2  Exam3 Letter
0    john     23     46     35      F
1    mary     42     31     36      F
2     sam     58     22     55      F
3  oksana     81     88     79      P
4     tom     11     19     18      F
5   peter     55     64     69      P
6  larisa     81     78     52      P
```

```
grades1[:4]
Out[25]:
     Name  Exam1  Exam2  Exam3 Letter
0    john     23     46     35      F
1    mary     42     31     36      F
2    sam      58     22     55      F
3  oksana     81     88     79      P
```

Single columns are selectable using the column name.

```
grades1['Letter']
Out[26]:
0    F
1    F
2    F
3    P
4    F
5    P
6    P
Name: Letter, dtype: object

grades1[['Name','Exam1']]
Out[27]:
     Name  Exam1
0    john     23
1    mary     42
2    sam      58
3  oksana     81
4    tom      11
5   peter     55
6  larisa     81
```

Now we consider descriptive statistics for numerical data. We will use the mean, standard deviation (std), variance (var), and percentile functions. For example, consider the variable Exam1 in the data grades.

```
exam1=grades1['Exam1']

np.mean(exam1)

Out[65]: 50.142857142857146

np.std(exam1)

Out[66]: 24.85632182552112

np.var(exam1)

Out[67]: 617.8367346938776


np.percentile(exam1,q=[25,50,75])

Out[68]: array([ 32.5,  55. ,  69.5])


np.median(exam1)

Out[69]: 55.0
```

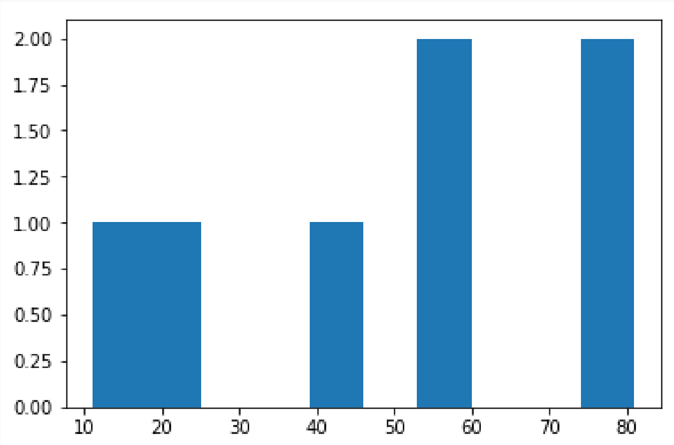
One way to find the mode of a data set is to use scipy.stats.

```
from scipy import stats

stats.mode(exam1)

Out[70]: ModeResult(mode=array([81], dtype=int64), count=array([2]))
```

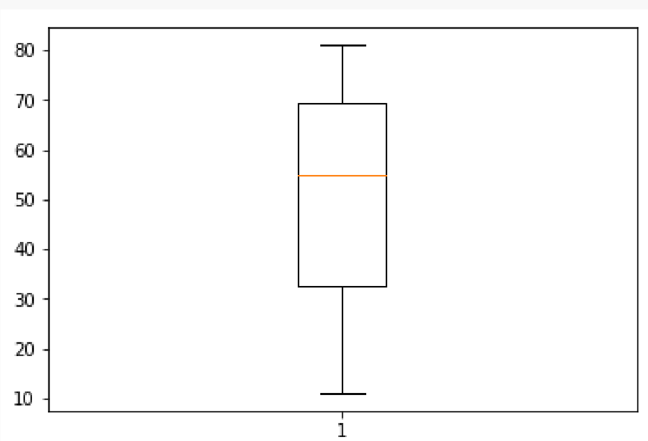
Matplotlib provides different plots to interact with the user.

```
import matplotlib.pyplot as plt

plt.hist(exam1)

plt.hist(exam1)

Out[72]:

(array([ 1.,  1.,  0.,  0.,  1.,  0.,  2.,  0.,  0.,  2.]),

 array([ 11.,  18.,  25., ...,  67.,  74.,  81.]),
```

```
 <a list of 10 Patch objects>)
```




```
plt.boxplot(exam1,sym='*')
Out[80]:
{'boxes': [<matplotlib.lines.Line2D at 0x2295a0ab710>],
 'caps': [<matplotlib.lines.Line2D at 0x2295a0b4a90>,
  <matplotlib.lines.Line2D at 0x2295a0bb940>],
 'fliers': [<matplotlib.lines.Line2D at 0x2295a0c39b0>],
 'means': [],
 'medians': [<matplotlib.lines.Line2D at 0x2295a0bbb00>],
 'whiskers': [<matplotlib.lines.Line2D at 0x2295a0ab8d0>,
  <matplotlib.lines.Line2D at 0x2295a0b48d0>]}
```



(2) Excel files, both 97/2003 (xls) and 2007/10/13 (xlsx), can be imported using read_excel in the package pandas. Two inputs are required to use read_excel, the filename and the sheet name containing the data. In this example, pandas makes use of the information in the Excel workbook that the first column contains dates and converts these to datetimes. Like the mixed CSV data, the array returned has object data type.

```
from pandas import read_excel

grades2= read_excel('grades.xlsx','Sheet1')#add path if needed

grades2

Out[82]:

     Name  Exam1  Exam2  Exam3 Letter

0    john     23     46     35      F

1    mary     42     31     36      F

2    sam      58     22     55      F

3  oksana     81     88     79      P

4    tom      11     19     18      F

5   peter     55     64     69      P

6  larisa     81     78     52      P
```

Functions

A function is a part of a program. It takes a list of argument values, performs a computation with those values, and returns a single result. Python gives you many built--in functions. But you can also create your own functions. These functions are called user-defined functions.

Functions are declared using the **def** keyword, and the value produced is returned using the **return** keyword. Consider a simple function which returns the square of the input, $y = x^2$.

```
def square(x):
    return x**2
# Call the function
x = 2
y = square(x)
print(x,y)
2 4
```

Function can also be defined using NumPy arrays and matrices.

```
import numpy as np
def l2_norm(x,y):
  d=x-y
  return np.sqrt(np.dot(d,d))
# Call the function
x = np.random.randn(10)
y = np.random.randn(10)
z = l2_norm(x,y)
print(x,y)
print("The L2 distance is ",z)


[-1.7552226  -1.13615929  0.08651004 -1.51614894  0.9905689   1.13026798
 -1.77935161 -0.30365381  0.62036808  0.66026817] [-0.52640001  0.83418918
0.27142506  1.08401711  1.49757908 -0.44359129
 -1.87576716 -0.74894177  0.74487044  1.17784902]
The L2 distance is  3.92586238858
```

Flow Control and Loops

Python uses white space changes to indicate the start and end of flow control blocks, and so **indention matters**. For example, when using if . . . elif . . . else blocks, all of the control blocks must have the same indentation level and all of the statements inside the control blocks should have the same level of indentation.

1. if . . . elif . . . else

if . . . elif . . . else blocks always begin with an if statement immediately followed by a *scalar* logical expression. elif and else are optional and can always be replicated using nested if statements at the expense of more complex logic and deeper nesting. The generic formof an if . . . elif . . . else block is

if *logical_1*:
   Code to run if logical_1
elif *logical_2*:
   Code to run if logical_2 and not logical_1
elif *logical_3*:
   Code to run if logical_3 and not logical_1 or logical_2
...
...
else:
   Code to run if all previous logicals are false

However, simpler forms are common:

if *logical*: *Code to run if logical true*

or

Code to run if logical true if *logical* else *Code to run if logical false*

```
x = 5
if x<5:
    x+=1
else:
    x-=1
x
Out[5]: 4


x = 5;
if x<5:
    x=x+1
```

```
elif x>5:
    x=x-1
else:
    x=x*2
 x
Out[10]: 10
```

2. for loop

for loops begin with for *item* in *iterable*:, and the generic structure of a for loop is
for *item* in *iterable*:
Code to run

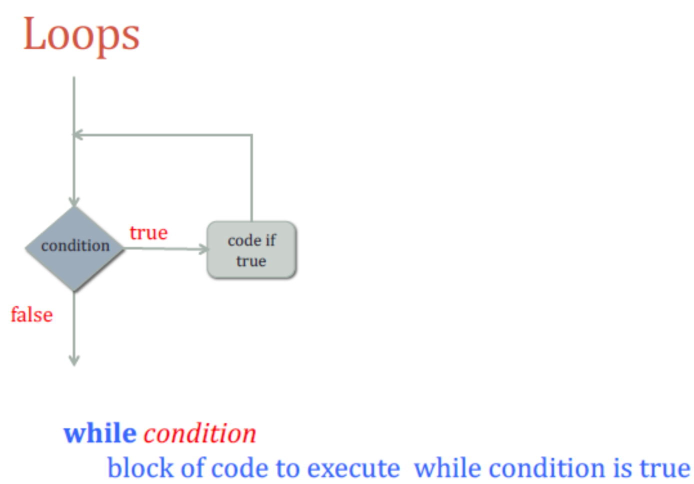
item is an iteration variable from *iterable*, and *iterable* can be anything that is iterable in Python. The most common examples are the built-in function range, lists, tuples, arrays or matrices. The for loop will iterate across all items in iterable, **beginning with item 0** and continuing until the final item.

The range() function allows you to iterate over a sequence of numbers.

```
for i in range(4):
 print(i)


0
1
2
3


for i in range(1, 10, 2):
 print(i)


1
3
5
7
```

```
9
```

3. While loop




```
count = 0
i = 1
while i<10:
  count += i
  i += 1
count
Out[13]: 45


count=0;
for i in range(0,10):
  count += i
count
Out[15]: 45
```

while loops should generally be avoided when for loops are sufficient. However, there are situations, for example the number of iterations required is not known in advance, where no for loop equivalent exists.

Discrete Probability Distributions

http://www.scipy-lectures.org/packages/statistics/index.html

https://docs.scipy.org/doc/scipy/

Scipy, and Numpy, provide a host of functions for performing statistical calculations. In this handout, we see how the **Scipy** package is used to calculate probabilities of a discrete random variable and random number generations.

Scipy is built on top of Numpy and uses Numpy arrays and data types. Hence we can use all the array manipulation and indexing methods provided by Numpy. Scipy imports all functions in the Numpy package, and several commonly used functions from sub-packages, into the top level namespace. For example, the Numpy array function is available as scipy.array. Similarly, the function scipy.var is the same as numpy.var. This means that we don't have to explicitly import Numpy.

```
import scipy as sp
import numpy as np
from scipy import stats

import matplotlib as mpl
from matplotlib import pyplot as plt
```

Part I. Probability mass function (PMF) and Cumulative distribution function (CDF)

(1) For discrete random variables, the probability mass function (PMF) gives the point probability of the variable having a specific value x_0.

(2) A **cumulative probability** of a random variable X (regardless of continuous or discrete) is the probability of that X is less than or equal to a specified value x_0, denoted by $F(x_0)=P(X \le x_0)$.

1. Binomial Distributions

```
from scipy.stats import binom
```

(1) binom.pmf(x, n, p)

(2) binom.cdf(x, n, p)

Example 1. $X \sim binomial\ (n=12, p=0.67)$. Find P($X$ = 5).

```
binom.pmf(5, 12, 0.67)
Out[17]: 0.045571862068520368
```

Example 2. $X \sim binomial\ (n=12, p=0.67)$. Find P($X \le 5$).

```
binom.cdf(5, 12, 0.67)
Out[18]: 0.063167533015853719
```

2. Geometric Distributions

```
from scipy.stats import geom
```

(1) geom.pmf(x, p)

(2) geom.cdf(x, p)

Example 1. $X \sim geometric\ (p=0.02)$. Find P($X$ = 5).

```
geom.pmf(5, 0.02)
Out[21]: 0.018447363199999997
```

Example 2. $X \sim geometric\ (p=0.02)$. Find P($X \le 2$).

```
geom.cdf(2, 0.02)
Out[18]: 0.039600000000000003
```

3. Negative Binomial Distributions

Note that the negative binomial random variable in Python is defined as the **number of failures** before the rth success.

```
from scipy.stats import nbinom
```

(1) nbinom.pmf(x, r, p)
(2) nbinom.cdf(x, r, p)

Example 1. *X ~ negative binomial (p=0.2, r=3),* where *X* is *the number of trials*. Find P(*X* = *7*).

```
nbinom.pmf(7-3, 3, 0.2)
Out[21]: 0.049152000000000015
```

Example 2. *X ~ negative binomial (p=0.2, r=3),* where *X* is *the number of trials*. Find P(*X* ≤ *7*).

```
nbinom.cdf(7-3, 3, 0.2)
Out[28]: 0.148032
```

4. Hypergeometric Distributions

$$p(k, M, n, N) = \frac{\binom{n}{k}\binom{M-n}{N-k}}{\binom{M}{N}}$$

```
from scipy.stats import hypergeom
```

(1) hypergeom.pmf(k, M, n, N)
(2) hypergeom.cdf(k, M, n, N)

Example 1. *X ~ Hypermetric (M=11, N=3, n=4).* Find P(*X* = *1*).

```
hypergeom.pmf(1, 11,4, 3)
Out[31]: 0.50909090909090893
```

Example 2. *X ~ Hypermetric (M=11, N=3, n=4)*. Find P(*X* ≤ *2*).

```
hypergeom.cdf(2, 11, 4, 3)
Out[33]: 0.97575757575757571
```

5. Poisson Distributions

```
from scipy.stats import poisson
```

(3) poisson.pmf(x, mu)
(4) poisson.cdf(x, mu)

Example 1. *X ~ Poisson (1)*. Find P(*X* = *2*).

```
poisson.pmf(2,1)
Out[35]: 0.18393972058572114
```

Example 2. *X ~ Poisson (1)*. Find P(*X* = *2*).. Find P(*X* ≥ *2*).

```
1-poisson.cdf(2-1, 1)
Out[36]: 0.26424111765711533
```

Part II. Random number generation

Random number generations can be done using the "rvs" function in Python.

Example. Generate 1000 Possion(μ=1) numbers.

```
from scipy.stats import poisson
r=poisson.rvs(mu=1, size=1000);
print(r);
```

Continuous Probability Distributions

http://www.scipy-lectures.org/packages/statistics/index.html

https://docs.scipy.org/doc/scipy/tutorial/stats/continuous.html

Scipy is built on top of Numpy and uses Numpy arrays and data types. Hence we can use all the array manipulation and indexing methods provided by Numpy. Scipy imports all functions in the Numpy package, and several commonly used functions from sub-packages, into the top level namespace. For example, the Numpy array function is available as scipy.array. Similarly, the function scipy.var is the same as numpy.var. This means that we don't have to explicitly import Numpy.

```
import scipy as sp
import numpy as np
from scipy import stats
```

Probability density function (PDF), Cumulative distribution function (CDF) and random number generation

(1) For continuous random variables, the probability density function (PDF) gives the value of a density function at a specific point x_0.

(2) A **cumulative probability** of a random variable X (regardless of continuous or discrete) is the probability of that X is less than or equal to a specified value x_0, denoted by $F(x_0)=P(X\leq x_0)$. See Handout 4.

(3) Random number generations can be done using the "rvs" function in Python.

1. Uniform Distributions

This distribution is constant between loc and loc + scale. By default, loc=0 and scale=1.

```
from scipy.stats import uniform
```

Example.

```
uniform.pdf(0.1)
Out[4]: 1.0
uniform.pdf(1.6, loc=1.5, scale=0.5)
Out[5]: 2.0
r = uniform.rvs(size=1000)
```

2. Normal Distributions

This distribution is determined by loc (mean parameter) and scale(standard deviation). By default, loc=0 and scale=1.

```
from scipy.stats import norm
```

Example. Consider the standard normal distribution

```
norm.pdf(x=0, loc=0, scale=1)
Out[8]: 0.3989422804014327
norm.cdf(x=0, loc=0, scale=1)
Out[9]: 0.5
norm.ppf(q=0.025, loc=0, scale=1) #Find a normal quantile
Out[10]: -1.9599639845400545
norm.rvs(loc=0, scale=1, size=100)
```

3. Exponential Distributions

A common parameterization for expon is in terms of the rate parameter lambda, such that pdf = (1/lambda) * exp(- *x/lambda*). This parameterization corresponds to using scale = lambda.

```
from scipy.stats import expon
```

Example. *X ~ exponential (lambda=2).*

```
expon.pdf(x=2, loc=0, scale=2)
Out[13]: 0.18393972058572117
expon.cdf(x=2, loc=0, scale=2)
Out[14]: 0.6321205588285577
expon.ppf(q=0.95, loc=0, scale=2)
Out[15]: 5.99146454710798
expon.rvs(loc=0, scale=2, size=100)
```

4. Gamma Distributions

The probability density function for gamma is:

$$f(x, a) = \frac{x^{a-1} \exp(-x)}{\gamma(a)}$$

for $x \geq 0$, $a > 0$. Here $\gamma(a)$ refers to the gamma function.

gamma has a shape parameter *a* which needs to be set explicitly.

When a is an integer, γ reduces to the Erlang distribution, and when $a = 1$ to the exponential distribution.

The probability density above is defined in the "standardized" form. To shift and/or scale the distribution use the `loc` and `scale` parameters. Specifically, `gamma.pdf(x, a, loc, scale)` is identically equivalent to `gamma.pdf(y, a) / scale` with `y = (x - loc) / scale`.

```
from scipy.stats import gamma
```

Example. *X ~ Gamma (α=3, β=2)* as in the textbook.

```
gamma.pdf(x=2, a=3, loc=0, scale=2)
Out[20]: 0.09196986029286057
gamma.cdf(x=0.5, a=3, loc=0, scale=2)
Out[21]: 0.002161496689762513
gamma.rvs(a=3, loc=0, scale=2, size=100)
```

5. Beta Distributions

```
from scipy.stats import beta
```

Example. *X ~ Beta (α=1, β=2).*

```
beta.pdf(x=0.5, a=1, b=2)
Out[26]: 1.0
beta.cdf(x=0.5, a=1, b=2)
Out[27]: 0.75
beta.rvs(a=1, b=2, size=100)
```