Applied Statistical Methods

Introduction to Python - Part I

Xuemao Zhang
East Stroudsburg University

January 20, 2023

Outline

- Python variables
- Python data types
  - Numeric Type
    - ★ Number
  - Boolean Type
    - ★ Boolean
  - Sequence types
    - ★ String
    - ★ List
    - ★ Tuple
  - Set Type
    - ★ Set
  - Mapping Type
    - ★ Dictionary
  - Arrays and Matrices in library `numpy`
- Python classes

Python variables

- You can enter commands one at a time at the command prompt (>>>) in IDLE. For example,

```
3+5
```

```
## 8
```

The above is the form of code in the lecture slides: the first line is the script(s) I typed; the second line is the result.

Note: # is the comment symbol in Python.

When you run the code 3+5 in your IDEL, you type after the command prompt >>> and it will look like this:

```
>>> 3+5
8
```

Python variables

- Python has no command for declaring a variable. A variable is created the moment you first assign a value to it.
- Variable names are case-sensitive.

```
a = 4
A = "Sally"
print(a)
```

```
## 4
```

```
print(A)
```

```
## Sally
```

- A variable can have a short name (like x and y) or a more descriptive name (age, carname, total_volume).

Python variables

- Rules for Python variables:
  - A variable name must start with a letter or the underscore character
  - A variable name cannot start with a number
  - A variable name can only contain alpha-numeric characters and underscores (A-z, 0-9, and _)
  - Variable names are case-sensitive (age, Age and AGE are three different variables)
- Legal variable names:

```
myvar = "John"
my_var = "John"
_my_var = "John"
myVar = "John"
MYVAR = "John"
myvar2 = "John"
```

- Illegal variable names:

```
2myvar = "John"
my-var = "John"
my var = "John"
```

Python variables

- Many Values to Multiple Variables: Python allows you to assign values to multiple variables in one line:

```
x, y, z = 1,2,3
print(x)
```

```
## 1
```

```
print(y)
```

```
## 2
```

```
print(z)
```

```
## 3
```

- And you can assign the same value to multiple variables in one line

```
x= y= z = 4
print(x)
```

```
## 4
```

```
print(y)
```

Output Variables

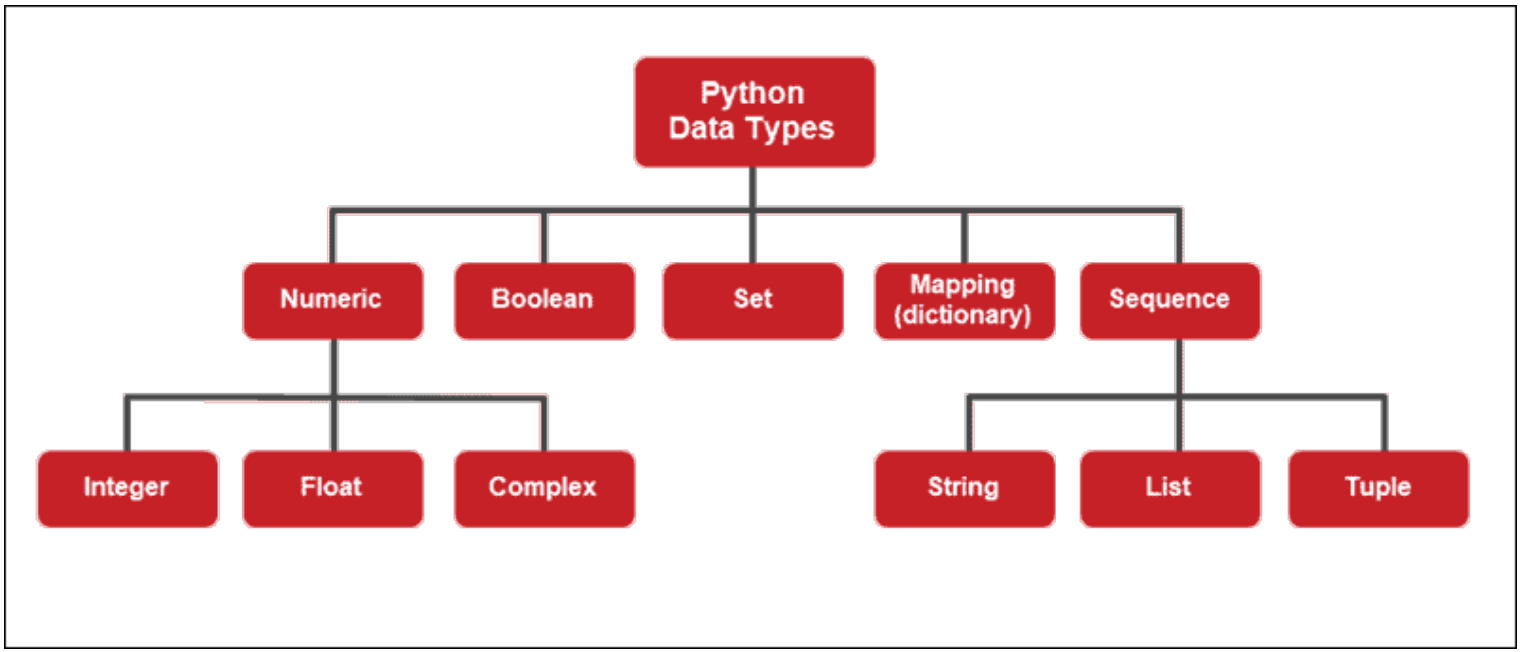
- The Python `print()` function is often used to output variables.
- In the `print()` function, you output multiple variables, separated by a comma:

```
print(x, y, z)
```

```
## 4 4 4
```

Python data types

- A **data type** is a characteristic that tells the compiler (or interpreter) how a programmer intends to use the data. There are two general categories of data types, differing whether the data is changeable after definition:
  - **Mutable**. Data types that are changeable after assignment.
  - **Immutable**. Data types that are not changeable after assignment.



Python data types: Number

- Python numbers are created by the standard way

```
var = 382
print(var)
```

```
## 382
```

- We use function `type()` to check data type

```
type(var)
```

```
## <class 'int'>
```

Python data types: Number

Type	Format	Description
int	a = 10	Signed Integer
float	a = 45.67	Floating point real values, numbers defined with a decimal point
complex	a = 1+2j	<real part> + <complex part>j

- 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 a function.

Python data types: Number

- For example, let's use the function `fload()` and `int()`
  - Note: Python3.x Version deleted the long integer data type and the `long()` function.

```
print(type(float(34)))
```

```
## <class 'float'>
```

```
print(type(int(34.5)))
```

```
## <class 'int'>
```

```
print(type(30))
```

```
## <class 'int'>
```

```
print(type(30/3))
```

```
## <class 'float'>
```

```
print( type(int(30/3)) )
```

```
## <class 'int'>
```

Python data types: Boolean

- Booleans represent one of two values: `True` or `False`.

```
print(type(True))
```

```
## <class 'bool'>
```

```
print(type(False))
```

```
## <class 'bool'>
```

```
print(True == 1)
```

```
## True
```

```
print(False == 0)
```

```
## True
```

```
print(2 == 2)
```

```
## True
```

```
print(2 == 3)
```

```
## False
```

Python data types: String

- **Sequence types** help represent sequential data stored into a single variable. There are three types of sequences used in Python
  - String
  - List
  - Tuple
- Strings are a sequence of bytes representing Unicode characters.
- Depending on the use case and characters needed, there are four different ways to create a string. They differ based on the delimiters and whether a string is single or multiline.

1. Create a string by using **double quote delimiters**

```
print("This is a string with 'single' quotes delimited by double quotes")
```

```
## This is a string with 'single' quotes delimited by double quotes
```

Python data types: String

2. Create a string by using **single quote delimiters**

```
print('This is a string with "double" quotes delimited by single quotes')
```

```
## This is a string with "double" quotes delimited by single quotes
```

3. Create a multiline string by using **triple single quote delimiters**

```
print("""'This is a multiline string
with 'single', "double" and ```triple single``` quotes
delimited with triple double quotes'""")
```

```
## 'This is a multiline string
## with 'single', "double" and ```triple single``` quotes
## delimited with triple double quotes'
```

4. Create a multiline string by using **triple double quote delimiters**

```
print("""'This is a multiline string
with 'single', "double" and ```triple single``` quotes
delimited with triple double quotes'""")
```

```
## 'This is a multiline string
## with 'single', "double" and ```triple single``` quotes
## delimited with triple double quotes'
```

Python data types: List

- A Python list is an **ordered mutable/changeable** array. Lists **allow duplicate elements** regardless of their type. Adding or removing members from a list allows changes after creation.
- Create a list in Python by using square brackets, separating individual elements with a comma.
  - Each element can be of any data type.
- 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.

```
A = [1, 2, "Bob", 3.4]
print(A, "is", type(A))
```

```
## [1, 2, 'Bob', 3.4] is <class 'list'>
```

Python data types: List

```
B = len(A)
# `len` will return the length of the list which is 3.
# The index is 0, 1, 2, 3.
print(A[0],A[1],A[2],A[3])
```

```
## 1 2 Bob 3.4
```

```
print(A[0:2]) #stop=2; the end element is not included
```

```
## [1, 2]
```

- The **colon operator** : is used for slicing, indexing a specific range and displaying the output using colon operator.
- By leaving out the start value, the range will start at the first item:

```
print(A[:2])
```

```
## [1, 2]
```

Python data types: List

- Negative indexes are also possible
  - -1 refers to the last item, -2 refers to the second last item etc.
  - Negative indexing is especially useful for navigating to the end of a long list of members.

```
print(A[-1])
```

```
## 3.4
```

Python data types: List

- To change the value of a specific item, refer to the index number

```
thislist = ["apple", "banana", "cherry"]
thislist[1] = 2
print(thislist)
```

```
## ['apple', 2, 'cherry']
```

- To change the value of items within a specific range, define a list with the new values, and refer to the range of index numbers where you want to insert the new values

```
thislist = ["apple", "banana", "cherry", "orange", "kiwi", "mango"]
thislist[1:3] = ["blackcurrant", "watermelon"]
print(thislist)
```

```
## ['apple', 'blackcurrant', 'watermelon', 'orange', 'kiwi', 'mango']
```

- If you insert more items than you replace, the new items will be inserted where you specified, and the remaining items will move accordingly

```
thislist = ["apple", "banana", "cherry"]
thislist[1:2] = ["blackcurrant", "watermelon"]
print(thislist)
```

```
## ['apple', 'blackcurrant', 'watermelon', 'cherry']
```

Python data types: List

- **Append** Items: To add an item to the end of the list, use the append() method

```
thislist = ["apple", "banana", "cherry"]
thislist.append("orange")
print(thislist)
```

```
## ['apple', 'banana', 'cherry', 'orange']
```

```
thislist.append(A) #append a list
print(thislist)
```

```
## ['apple', 'banana', 'cherry', 'orange', [1, 2, 'Bob', 3.4]]
```

- The elements of a list can be list

```
A=[[1,2],[3,4]]
B=[5,6]
A.append(B)
print(A)
```

```
## [[1, 2], [3, 4], [5, 6]]
```

Python data types: List

- **Extend** Items: To add all elements of an item to the end of the list, use the `extend()` method
- The following is to **append elements** from another list to the current list by the `extend()` method.

```
A=[[1,2],[3,4]]
B=[5,6]
A.extend(B)
print(A)
```

```
## [[1, 2], [3, 4], 5, 6]
```

Python data types: List

- Insert Items: To insert a list item at a specified index, use the `insert()` method. The `insert()` method inserts an item at the specified index
  - A list can be inserted

```
thislist = ["apple", "banana", "cherry"]
thislist.insert(2, "orange")
print(thislist)
```

```
## ['apple', 'banana', 'orange', 'cherry']
```

Python method

What is a **Python method**?

- Python is an **object-oriented programming** language. Objects in object-oriented programming have attributes, i.e. data values within them. But how do we access these attribute values? We access them through methods.
- **Methods** represent the **behavior of an Object**.
- A method is a piece of code that is associated with an object, operates upon the data of that object, and return an object.
- We call a method on an object using the **dot operator** .

Python data types: List

- Remove Specified Item: The `remove()` method removes the specified item.

```
thislist = ["apple", "banana", "cherry"]
thislist.remove("banana")
print(thislist)
```

```
## ['apple', 'cherry']
```

- If you do not specify the index, the `pop()` method removes the last item.

```
thislist = ["apple", "banana", "cherry"]
thislist.pop()
```

```
## 'cherry'
```

```
print(thislist)
```

```
## ['apple', 'banana']
```

Python data types: List

- The `del` keyword also removes the specified index

```
thislist = ["apple", "banana", "cherry"]
del thislist[0]
print(thislist)
```

```
## ['banana', 'cherry']
```

- The del keyword can also delete the list completely.

```
thislist = ["apple", "banana", "cherry"]
del thislist
# print(thislist)
```

- The `clear()` method empties the list. The list still remains, but it has no content.

```
thislist = ["apple", "banana", "cherry"]
thislist.clear()  # thislist[:]=[]
print(thislist)
```

```
## []
```

Python data types: List

- List objects have a sort() method that will sort the list alphanumerically, ascending, by default

```
thislist = ["orange", "mango", "kiwi", "pineapple", "banana"]
thislist.sort()
print(thislist)
```

```
## ['banana', 'kiwi', 'mango', 'orange', 'pineapple']
```

```
thislist = [100, 50, 65, 82, 23]
thislist.sort()
print(thislist)
```

```
## [23, 50, 65, 82, 100]
```

Python data types: List

- By default the sort() method is case sensitive, resulting in all capital letters being sorted before lower case letters

```
thislist = ["banana", "Orange", "Kiwi", "cherry"]
thislist.sort()
print(thislist)
```

```
## ['Kiwi', 'Orange', 'banana', 'cherry']
```

- Case Insensitive Sort: Luckily we can use built-in functions as key functions when sorting a list. So if you want a case-insensitive sort function, use str.lower as a key function

```
thislist = ["banana", "Orange", "Kiwi", "cherry"]
thislist.sort(key = str.lower)
print(thislist)
```

```
## ['banana', 'cherry', 'Kiwi', 'Orange']
```

Python data types: List

- To sort descending, use the keyword argument reverse = True

```
thislist = [100, 50, 65, 82, 23]
thislist.sort(reverse = True)
print(thislist)
```

```
## [100, 82, 65, 50, 23]
```

- Reverse Order: The `reverse()` method reverses the current sorting order of the elements.

```
thislist = [100, 50, 65, 82, 23]
thislist.reverse()
print(thislist)
```

```
## [23, 82, 65, 50, 100]
```

Python data types: List

- We have seen that lists aren't limited to a single dimension. You can declare multiple dimensions by separating them with commas. In the following example, the MyTable variable is a two-dimensionallist.

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

- Another example

```
list1 = [100, 50, 65, 82, 23]
list2= [1,2,3,4,5]
list3=[list1, list2]
print(list3)
```

```
## [[100, 50, 65, 82, 23], [1, 2, 3, 4, 5]]
```

Python data types: 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.
  - A tuple is a collection which is ordered and unchangeable, and allow duplicate values.
    - ★ Elements cannot be removed from a tuple. Tuples have no append or extend method.
  - Tuples are defined by parenthesis ().
  - 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.

```
x= ("bare", "metal", "cloud", 2.0, "cloud")
print(x)
```

```
## ('bare', 'metal', 'cloud', 2.0, 'cloud')
```

```
y=(2,3,4,5)
print(y)
```

```
## (2, 3, 4, 5)
```

```
z=(y, 6)  #two dimensional tuple
print(z)
```

```
## ((2, 3, 4, 5), 6)
```

Python data types: Set

- The Set data type is part of the set class. It stores data collections into a single variable.
  - Sets **do not allow duplicate values**.
  - Set items are **unchangeable**, but you can remove items and add new items.
  - Sets are **unordered** and unindexed, so you cannot be sure in which order the items will appear.
- To create a set, use the curly brackets notation and separate individual elements with a comma.

```
s1 = {1, 2, 3, 3, 3, 4}
print(len(s1))
```

```
## 4
```

```
print(s1)
```

```
## {1, 2, 3, 4}
```

```
s2 = {"abc", 34, True, 40, "male"}
print(s2)
```

```
## {True, 34, 'abc', 40, 'male'}
```

```
print(len(s2))
```

```
## 5
```

Python data types: Set

- It is also possible to use the `set()` constructor to make a set.

```
thisset = set(("apple", "banana", "cherry")) # note the double round-bracke
print(thisset)
```

```
## {'cherry', 'banana', 'apple'}
```

Python data types: dictionary

- **Mapping data type** is represented by a Python dictionary.
- A **dictionary** is a collection of data with **key and value pairs** belonging to the dict class.
- There are four collection data types in the Python programming language:
  - **List** is a collection which is ordered and changeable. Allows duplicate members.
  - **Tuple** is a collection which is ordered and unchangeable. Allows duplicate members.
  - **Set** is a collection which is unordered, unchangeable, and unindexed. No duplicate members. But you can remove items and add new items.
  - **Dictionary** is a collection which is ordered and changeable. No duplicate members.

Python data types: dictionary

- To create a dictionary, use the curly bracket notation and define the key value pairs.
  - Dictionaries are changeable, meaning that we can change, add or remove items after the dictionary has been created.
  - Duplicates Not Allowed. The keys must be unique.

```
d = {"cost":2.2,
     "articles":10,
     True:"Okay!",
     2:"One"}
print(d)
```

```
## {'cost': 2.2, 'articles': 10, True: 'Okay!', 2: 'One'}
```

```
print(type(d))
```

```
## <class 'dict'>
```

Python data types: dictionary

```
d['articles'] = 11  # set the value associated with the 'articles' k
print(d['cost']) # print the value of the 'cost' key
```

```
## 2.2
```

```
d['book'] = "Python" # Add a new key 'book' with the associated valu
print(d.keys()) # print out a list of keys in the dictionary
```

```
## dict_keys(['cost', 'articles', True, 2, 'book'])
```

- The `in` operator (see more in Lecture 5) determines whether a given value is a constituent element of a sequence such as a string, array, list, or tuple.

```
print ('cost' in d) # test to see if 'cost' is in the dictionary.
```

```
## True
```

Python data types: dictionary

- An `OrderedDict` is a dictionary subclass that **preserves the order** that keys were first inserted.
- The OrderedDict is a subclass of dict object in Python. The only difference between OrderedDict and dict is that, in OrderedDict, it maintains the orders of keys as inserted. In the dict, the ordering may or may not be happen. The OrderedDict is a standard library class, which is located in the `collections` module.
  - https://docs.python.org/3/library/collections.html

Python data types: dictionary

```
from collections import OrderedDict
students = OrderedDict()
students["Name"] = "Michelle"
students["ID"] = 19210293
students["Branch"] = "Computer Science"

print(students)
```

```
## OrderedDict([('Name', 'Michelle'), ('ID', 19210293), ('Branch', 'Computer Science
```

- OrderedDict class has the method move_to_end() defined specifically to modify the order of items.

```
students.move_to_end("ID")
print(students)
```

```
## OrderedDict([('Name', 'Michelle'), ('Branch', 'Computer Science'), ('ID', 192102
```

- More examples: https://pythongeeks.org/ordereddict-in-python/

Python data types: Array

- Defined in library numpy (http://www.scipy-lectures.org/), vectors and matrices are for numerical data manipulation.

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

```
## <class 'numpy.ndarray'>
```

```
print(arr1)
```

```
## [1 2 3 4 5]
```

```
print(arr1[1])
```

```
## 2
```

Python data types: Array

- Arrays can be two-dimensional

```
data2 = [range(1, 5), range(5, 9)] # list of lists
# range() generates a range of numbers
arr2 = np.array(data2) # 2d array or matrix.
# arr2 = np.array([range(1, 5), range(5, 9)])

print(arr2)
```

```
## [[1 2 3 4]
##  [5 6 7 8]]
```

```
print(np.shape(arr2)) #dimensions of the matrix
```

```
## (2, 4)
```

```
print(type(arr2))
```

```
## <class 'numpy.ndarray'>
```

```
print(arr2[0,1])
```

```
## 2
```

```
print(arr2[1,1])
```

```
## 6
```

Python data types: Array

```
print(arr2[0, :]) # row 0
```

```
## [1 2 3 4]
```

```
print(arr2[:, 0]) # column 0
```

```
## [1 5]
```

```
print(arr2[:, :2])
#columns strictly before index 2 (the first 2 columns)
```

```
## [[1 2]
##  [5 6]]
```

Python data types: Array

```
print(arr2[:, 2:])
# columns after index 2 ( column 2 included)
```

```
## [[3 4]
##  [7 8]]
```

```
print(arr2[:, 1:4])
# columns between index 1 (included) and 4 (excluded)
```

```
## [[2 3 4]
##  [6 7 8]]
```

Python data types: Array

- The double colon `::` operator in python are used for jumping of elements in multiple axes. It is also a slice operator. Every item of the sequence gets sliced using double colon.

```
print(arr2[:, 2::])  #it is equivalent to print(arr2[:, 2:])
```

```
## [[3 4]
##  [7 8]]
```

```
print(arr2[:, 2:])
```

```
## [[3 4]
##  [7 8]]
```

Python data types: Array

- The syntax of a Slice operator using double colon is [**Start** : **Stop** : **Steps**].
  - **Start** (Indicates the number from where the slicing will start),
  - **Stop**(Indicates the number where the slicing will stop) and
  - **Steps**(Indicates the number of jumps interpreter will take to slice the string) are the three flags and all these flags are integer values.

```
print(arr2[:, 0:3:2]) # all rows, every other column
#The above code can be reduced to a short cut by using double colon ::
```

```
## [[1 3]
##  [5 7]]
```

```
print(arr2[:, ::2])
```

```
## [[1 3]
##  [5 7]]
```

```
print(arr2[:, ::-1]) # reverse order of columns
```

```
## [[4 3 2 1]
##  [8 7 6 5]]
```

Python data types: Array

```
y=arr2[:, [0,1,2,2]] #column 0,1,2,2 of arr2
print(y)
```

```
## [[1 2 3 3]
##  [5 6 7 7]]
```

- vector and matrix of ones

```
x1=np.ones(3)
print(x1)
```

```
## [1. 1. 1.]
```

```
x2=np.ones((3,3))
print(x2)
```

```
## [[1. 1. 1.]
##  [1. 1. 1.]
##  [1. 1. 1.]]
```

Python data types: Array

- vector and matrix of zeros

```
x1=np.zeros(3)
print(x1)
```

```
## [0. 0. 0.]
```

```
x2=np.zeros((3,3))
print(x2)
```

```
## [[0. 0. 0.]
##  [0. 0. 0.]
##  [0. 0. 0.]]
```

```
x2[0,0]=999  #change element
print(x2)
```

```
## [[999.   0.   0.]
##  [  0.   0.   0.]
##  [  0.   0.   0.]]
```

Python data types: Matrix

- Matrices in `numpy` are similar to arrays but they can only have two dimensions.
  - Arrays can be of any dimensions.
  - message from numpy doc: `It is no longer recommended to use this class, even for linear algebra. Instead use regular arrays. The class may be removed in the future.`

```
A=np.matrix([[1,2,3],[4,5,6],[7,8,9]])
print(A)
```

```
## [[1 2 3]
##  [4 5 6]
##  [7 8 9]]
```

```
print(type(A))
```

```
## <class 'numpy.matrix'>
```

Python data types: Matrix

```
A=np.array([[1,2,3],[4,5,6],[7,8,9]], dtype=float)
print(A)
```

```
## [[1. 2. 3.]
##  [4. 5. 6.]
##  [7. 8. 9.]]
```

```
print(type(A))
```

```
## <class 'numpy.ndarray'>
```

- We check the mathematical functions and operators for arrays and matrices in next lecture.

Python classes

- Python classes provide a means of bundling data and functionality together. Creating a new class creates a new type of **object**, allowing new **instances** of that type to be made. Each class instance can have **attributes** attached to it for maintaining its state. Class instances can also have **methods** (defined by its class) for modifying its state.
- https://docs.python.org/3/tutorial/classes.html
- There is no way to avoid python classes, but we do not need to build our own classes in this course.

License