

# BST 261: Data Science II

Heather Mattie

Department of Biostatistics  
Harvard T.H. Chan School of Public Health  
Harvard University

March 25, 2019

# Python Review

# Why learn Python?

- Python is a clear and powerful programming language
- Elegant syntax makes programs easy to read
- Large standard library supports many common programming tasks
- Python can be used in two different modes
  - **Interactive mode** makes it easy to experiment with the language
  - **Standard mode** is for running scripts and programs from start to finish
- Python programs are much shorter than equivalent C, C++, or Java programs
- High-level language and data types allow concise complex operations

# Which Python to learn?

- In addition to core Python, will also use several Python packages to perform scientific computations
- Instead of installing Python packages one at a time, we will use **Python distribution**
- There are currently several Python distributions available, will use Anaconda Python Distribution
- In October 2015, the distribution for Python 3.4 supported 274 packages
- In October 2016, the distribution for Python 3.5 supported 445 packages
- In March 2017, the distribution for Python 3.6 supports 452 packages
- Anaconda includes development environments Jupyter (formerly, IPython Notebook) and Spyder

- Some like Python 2.x and some like Python 3.x
- What are the differences?
  - Short version: Python 2.x is legacy, Python 3.x is the present and future of the language ([www.python.org](http://www.python.org))
  - The final 2.x version 2.7 release came out in mid-2010 (no further major releases)
  - Python 3.0 was released in 2008, Python 3.1 in 2009, Python 3.2 in 2011, Python 3.3 in 2012, Python 3.4 in 2014, Python 3.5 in 2015, and Python 3.6 in 2016

- Downside: breaking backwards compatibility in 3.x means that some software, especially in-house software in companies, still doesn't work on 3.x
- Guido van Rossum, the original creator of the Python language, decided to clean up Python 2.x with less regard for backwards compatibility
- All recent standard library improvements only available by default in Python 3.x
- Python 3.x is easier for newcomers to learn and several aspects of the core language are more consistent
- Eliminates many quirks that can trip up beginning programmers learning Python 2 (e.g., division by integers)

# Basic Data Types

- Python contains many data types as part of the core language, such as numbers
- All data in a Python program is represented by objects or by relations between objects
- The value of some objects can change. Objects whose value can change are said to be **mutable**, whereas objects whose value is unchangeable once they are created are called **immutable**
- Python also contains built-in functions that can be used by all Python programs
- The “Python library” consists of all core elements, such as data types and built-in functions, but the bulk of it consists of modules
- To make use of modules in your code, you first need to import them using the `import` statement



```
1 # print object
2 print("hello world")
3
4 # return the largest item
5 max(2,9,5)
6
7 # return a number rounded to n digits after the decimal point (floating point)
8 round(12.345, 1)
9
10 # return a number rounded to the nearest integer (int)
11 round(12.345)
12
13 # return a new sorted list from the items in iterable
14 sorted([4,2,3,1])
15
16 # logical operations
17 2 < 3 < 4
```

# Object Types

- Every piece of data stored in a Python program is an object
- Each object has a **type**, **value**, and **identity**
- For example: `str` (type); `"hello"` (value); 2915232 (identity)
- **Attribute** is a name that is attached to a specific object and has a dot between the two: `object.attribute`
- Some attributes are callable (functions) and others are noncallable (variables)
- Objects are characterized by data attributes and methods
  - A **data attribute** is a value attached to an object
  - A **method** is a function attached to an object that performs some operation on the object
- Object type determines which operations the object supports, i.e., which operations you can perform on it
- An instance is an occurrence of an object, and different instances share the same set of attributes

```
1 import numpy as np
2 x = np.array([2,4,6,8])
3 y = np.array([1,3,9])
4 x.mean()
5 y.mean()
6 x.shape
7 y.shape
```

- Python **modules** are libraries of code
- They are imported using the `import` statement
- Python comes with several modules but you can also write your own modules
- For example, mathematical functions are available in the `math` module

```
1 from math import pi, sqrt
2 import math
3 math.pi
4 math.e
5 math.sqrt(10)
6 math.sin(math.pi/2)
```

- As programs grow in size, it will be useful to break them into multiple files for easier maintenance
- Large Python programs are usually organized into modules and then loaded using the `import` statement
- The `import` function does three things:
  - Creates a new namespace for all the objects defined in the source file
  - Executes the code in the module within the newly created namespace
  - Creates a name, which matches the name of the module, within the caller that refers to the module namespace
- Subsequent calls to `import` neither load nor execute the code, but they will bind the module name to the module object created earlier
- A namespace is space or container of names to prevent naming conflicts

```
1 import numpy as np
2 import math
3 math.sqrt(2) # math.sqrt takes in a single number and returns a number
4 np.sqrt([2,3]) # np.sqrt takes in either a number or a sequence and returns an
    np.array object
```

# Object Types

- To find out the type of an object:

```
1 name = "Amy Poehler"
```

- To find out a list of all attributes (object.attribute):

```
1 dir(name)
```

- Note that some of the attributes are callable (methods)
- To find out what different methods do:

```
1 help(name.lower)    # documentation on the string-function "lower"  
2 help(name.lower())  # documentation on "amy meredith poehler" (!)
```

- Note that the above examples use names (object instances), but one can also use object types

```
1 help(str)
```

- Numbers are one type of object in Python
- Python provides 3 numeric types: integers, floating-point numbers, and complex numbers
- Integers have unlimited precision
- Different numeric types can be freely mixed
- Numbers support all the usual arithmetic operations

```
1 125 + 25.21
2 125 + 25
3 125 * 25
4 125 ** 25
5 5 / 2      # normal division
6 5 // 2     # floor division
7 1 + _
```

- In the interactive mode, the `_` operator contains the result of the last operation (very handy!)

- Very commonly need to go beyond built-in functions and operations
- We can import the `math` module for this purpose

```
1 import math
2 math.sqrt(math.pi)
3 math.sin(math.sqrt(math.pi)/2)
```

- `random` is a useful module when dealing with simple settings that require random sampling of a sequence of objects

```
1 import random
2 random.random()
3 random.choice([10, 15, 20, 25, 30, 35, 40])
4 random.choice(["Monday", "Wednesday", "Friday"])
```

- **Expression is a combination of objects and operators that computes a value**
- Many expressions involve what is known as the Boolean data type
- The Boolean data type is a data type that has only two values: true and false
- In Python, the Boolean type is `bool`, and it has two values: `True`, `False`
- Compare this to the `int` type which can represent any integer
- Operations involving logic, so-called Boolean operations, take in one or more Boolean objects and return one Boolean object
- There are only 3 Boolean operations (`x` and `y` are Boolean; listed here by ascending priority)

```
1 x or y
2
3 x and y
4
5 not x
```



# Comparisons

- There are 8 comparison operations in Python
- These are commonly used for numeric types
- Can also be used for other types, like sequences, where comparisons are done element wise
- The result of a comparison is either `True`, `False` (returns a Boolean type)
- All have the same priority (higher than Boolean)
- Here `x` and `y` are any objects for which the comparison operators are defined:

```
1 x < y
2 x <= y
3 x > y
4 x >= y
5 x == y
6 x != y
7 is
8 is not
```

- The last two compare object identity and its negation
- We can also chain comparisons in Python

```
1 2 < 3 < 4
```

# Sequences

- In Python, a **sequence** is a collection of objects ordered by **position**
- There are 3 basic sequence types: lists, tuples, and range objects
- Additional sequence types exist for representing strings (text)
- All sequences support **common sequence operations** and each sequence type has its **additional type specific operations**
- These data types are called sequences because the objects they contain form a sequence



- Indexing for sequences starts from zero (0)
- Indexing logic: [from position, to position)

- **Strings are immutable sequences of characters**
- String literals are enclosed in single ('hi '), double("hi "), or triple quotes ("""Computer says "No.""")
- Examples of common sequence operations on strings

```
1 S = "Python"
2 len(S)
3 S[0]
4 S[0:6] # slicing
5 S[-1]
6 "y" in S # membership
7 "z" not in S
```

- **Polymorphism** is the idea that the meaning of an operation, such as + and \*, depends on the objects being operated on
- This is like in math: each operation has to be defined separately for each type of object

```
1 S = "Python"
2 3*5
3 3*S
4 3 + 5
5 S + ' is phun'
6 print("eight is " + 8)          # string object + integer object (!)
7 print("eight is " + str(8))    # string object + string object
```

- Here + is referred to as “concatenation” and \* as “repetition”

- The above operations on strings were really generic sequence operations
- Strings are a type of sequence and support all sequence operations
- In addition, strings have operations all their own, which are available as **string methods**
- Methods are functions attached to objects and they are triggered with a call expression

```
1 S = 'Python'
2 S.find('y')
3 S.replace('y', 'Y')
4 name = 'Amy Poehler'
5 name.split(' ')
6 print(name)
7 name.upper()
8 print(name)
9 name.lower()
10 print(name)
```

- **Lists are mutable sequences of objects of any type** typically used to store homogeneous items
- Lists are a type of sequence
- Strings vs. lists: Sequences of characters vs. sequences of any objects
- Strings vs. lists: Strings are immutable whereas lists are mutable
- It is common in practice for lists to hold objects of one type only (which may consist of many kinds of nested objects)

```
1 names = ["Peter", "John", "Mary"]
2 names[0]
3 names[1]
4 names.append("Tom")
5 names = names + ["Jim"]
6 import random
7 random.choice(names)
```

```
1 names.reverse() # reverses the list in place (common mutable sequence operation)
2 names.sort() # sorts the list in place (list method)
```

- Because lists are mutable, the above method calls actually modify the original list
- Because strings are immutable, they cannot modify the string itself, so many methods return a new string
- Therefore, with strings, if you want to store the result, you need to assign it to a variable

```
1 name = "Juan Pedro"
2 name.upper()
3 print(name)
4 name = name.upper()
5 print(name)
6 numbers = [1,5,4,7,9,5,3,4,5,6,5,3,4,5,6,7,5,3,2,3,4,5,6,5,3,2,3,1]
7 numbers.count(4) # common sequence operation
```

- **Tuples are immutable sequences** typically used to store heterogeneous data
- Tuples are best viewed as a single object consisting of several parts
- You can use a tuple to return multiple values from a function (coming up)
- Although tuples and lists can often be used to perform similar tasks, creating short lists wastes memory because Python optimizes the performance of certain methods (such as `append()`) by slightly over-allocating memory

```
1 T = (1,2,3,4)
2 len(T)
3 T + (5,6)      # concatenation
4 T[0]
5 T[1] = 20      # immutable (!)
6 x = 12.3; y = 22.5
7 coordinates = (x,y)    # tuple packing
8 (x, y) = coordinates  # tuple unpacking
9 x, y = coordinates     # tuple unpacking
10 coordinates = [(x,x**2) for x in range(10)]
11 for x, y in coordinates:
12     print(x,y)
```



- **Ranges are immutable sequences of integers** commonly used in `for` loops

```
1 list(range(5)) # stop
2 list(range(1,6)) # start and stop
3 list(range(0,11,2)) # start and stop and step
```

- Can use negative steps but cannot use non-integer steps
- Note that `range` objects are different from `list` objects
- Although could use a list in `for` loops, `range` is much more memory efficient as it stores only start, stop, and step values, calculating individual items as needed

- **Sets are unordered collections of distinct hashable objects**
- There are two types of sets: `set` is mutable whereas `frozenset` is immutable
- A set is an unordered collection of objects and it cannot be indexed
- The elements of a set are never duplicated, i.e., adding the same element again to a set has no effect (cf. `list.append()`)
- Useful for keeping track of distinct objects and for doing mathematical set operations (union, intersection, set difference)
- Sets come with nonmutating and mutating methods, where the mutating methods can be called only on instances of type `set`

```
1 nodes = set()
2 nodes = set([2,4,6,8,10,12])
3 nodes.add(14)
4 nodes.add(6) # adding a single item at a time
5 nodes.update([6,14]) # item adding multiple items at a time
6 nodes.pop() # remove and return an arbitrary element
7 nodes.remove(4) # remove a specified element
8 females = set([2,6,8])
9 males = nodes.difference(females)
10 print(nodes) # difference is a nonmutating method
11 2 in males # testing set membership
```

- There are handy shortcuts for set operations:

```
1 males = nodes - females # set difference
2 everyone = males | females # set union
3 nobody = males & females # set intersection
```

- As a simple application, let's use sets to count the number of unique letters in a word

```
1 word = "an-ti-dis-es-tab-lish-ment-ar-i-an-ism" # syllabified
2 word = word.replace("-", "")
3 letters = set(word)
4 count = len(letters)
```

- There is no need to do this in steps (apart from clarity) if we don't need the intermediate results:

```
1 count = len(set("floc-ci-nau-ci-ni-hil-i-pil-i-fi-ca-tion".replace("-", "")))
```



- **Dictionaries are mappings from key objects to value objects**
- Consist of key-value pairs, where keys must be immutable (actually, hashable) and the values can be anything
- Dictionaries themselves are mutable, and can be seen as associative arrays
- Dictionaries can be used for performing fast lookups on unordered data
- Note that dictionaries are not sequences, and therefore do not maintain any type of left-to-right order, i.e., keys and values are iterated over in an arbitrary order
- Diagram of key-value pairs and lack of order

- Some examples of how to use dictionaries

```
1 md = {}  
2 md = dict()  
3 age = {'Tim': 29, 'Jim': 31, 'Pam': 27, 'Sam': 35}  
4 age['Tim']  
5 age['Tim'] = age['Tim'] + 1  
6 age['Tim'] += 1  
7 age['Tom'] = 45
```

- The objects returned by `dict.keys()`, `dict.values()` and `dict.items()` are **view objects**
- Provide a dynamic view on the dictionary's entries: when the dictionary changes, the view reflects these changes

```
1 age = {'Tim': 29, 'Jim': 31, 'Pam': 27, 'Sam': 35}
2 names = age.keys()
3 age['Tom'] = 45
4 print(names)
5 'Tim' in age
```

- To get a new view of the dictionary's keys

```
1 age.keys()
```

- To get a new view of the dictionary's values

```
1 age.values()
```

- Dictionary views support iteration and membership tests

```
1 "Tom" in names
2 age['Nick'] = 52
3 "Nick" in names
```

# Dynamic Typing



- Some languages are statically typed (e.g., C and C++) and others are dynamically typed (e.g., Python)
- Static typing means that type checking is performed during compile-time
- Dynamic typing means that type checking is performed at run-time
- What is type checking? The process of verifying and enforcing the constraints of types. If you move data from one variable to another, if the types of these variables do not match, you could potentially lose information. Typing, or assigning data types, refers to the set of rules that the language uses to ensure that the part of the program receiving some data knows how to correctly interpret that data.

- Consider the following simple example:

```
1 x = 3
```

- How does Python know that `x` should stand for an integer?
- Three important concepts: variable (name), object, and reference
- When assigning variables to objects, the following three things happen:

**Object:** Python creates an object to represent the value 3

**Variable:** Python creates the variable `x` (if it does not exist yet)

**Reference:** Python links the variable to the object

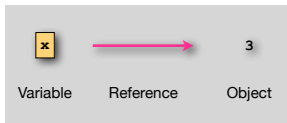


Figure: Schematic of dynamic typing.

# Dynamic Typing

- Variables and objects are stored in different parts of computer's memory
- **Variables (names) always link to objects, never to other variables**
- A variable thus becomes a reference to the given object, i.e., a name (possibly one out of many) attached to the object
- Larger objects may link to other objects
- Only objects have types:

```
1 x = 3
2 type(x)
3 x = "hello"
4 type(x)
```

- The variable `x` now references a different type of object (string, not integer)
- However, the type of `x` itself has not changed because `x` (the variable) has no type

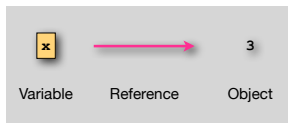


Figure: Schematic of dynamic typing.

- When assigning objects, it's useful to keep in mind the distinction between mutable and immutable objects
  - **Mutable objects** can be altered (lists, dictionaries)
  - **Immutable objects** cannot be altered (numbers, strings)
- Let's start with an easy example:

```
1 x = 3
2 y = x
```

- This may be confusing sometimes:

```
1 x = 3
2 y = x
3 x = x - 1
```

- What is the value of `x` and `y` after the last operation?

# Dynamic Typing

```
1 x = 3
2 y = x
3 x = x - 1
```

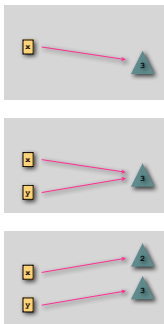


Figure: Schematic of the problem.

- The key to understanding this is to realize that **numbers are immutable objects**
- Consequently, the subtraction results in a **new object** being created
- For **immutable objects** assignment effectively creates a copy

- How about this example for a list?
- The behavior for **mutable** objects (e.g., lists, dictionaries) looks different, although it really follows the same logic

```
1 L1 = [2,3,4]
2 L2 = L1
3 L1[0] = 24
4 L1
5 L2
```

# Dynamic Typing

```
1 L1 = [2,3,4]
2 L2 = L1
3 L1[0] = 24
4 L1
5 L2
```

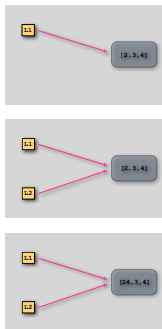


Figure: Schematic of the problem.

- Because the target object can be modified (lists are mutable), the operation does not result in the creation of a new object (cf. numbers)

# Dynamic Typing

- Each object has a **type**, **value**, and **identity**
- Mutable objects can be identical in content, and yet be different objects

```
1 L = [1,2,3]
2 M = [1,2,3]
3 L == M
```

True

```
1 L is M      # testing for object identity
```

False

- We can use the `id` function to obtain the identity of an object, which corresponds to its location in memory

```
1 id(L)
```

4316337288

```
1 id(M)
```

4312749840

```
1 id(L) == id(M)
```

False



- What if you wanted to copy a list, not just another reference (synonym) for it?
- The most elegant way:

```
1 L = [1,2,3,4,5,6,7]
2 M = list(L)
3 M is L
```

False

- Could also use slicing:

```
1 M = L[:]
```

- For more complex structures, there is also the `copy` module:

```
1 import copy
2 newlist = copy.copy(L)
3 newlist = copy.deepcopy(L)
```

- A **shallow copy** constructs a new compound object and then **inserts references** into it to the original
- A **deep copy** constructs a new compound object and then, recursively, **inserts copies** into it of the original objects

## Compound Statements

# Python Compound Statements

- Compound statements contain (groups of) other statements and they affect or control the execution of those other statements in some way
- Compound statements typically span multiple lines
- A compound statement consists of one or more **clauses** where a clause consists of a header and a block or suite of code
- The clause headers of a particular compound statement start with a keyword, end with a colon, and are all at the same indentation level
- **A block or suite of code of each clause however must be indented to indicate that it forms a group of statements that logically fall under the header**
- There are no hard-and-fast rules about indentation as long as you are consistent
- Tab and 4 spaces are probably the most common choices
- Here's an example of a compound statement with 1 clause (1 header line + 2 lines of code in the block)

```
1 if x > y:
2     difference = x - y
3     print("x is greater than y")
4     print("But this gets printed no matter what!")
```

# Python if Statement

- The Python if statement:

```
1 if test:
2     [block of code]
3 elif test:
4     [block of code]
5 else:
6     [block of code]
```

- The if statement selects from among one or more actions, and it **runs the block associated with the first if or elif test that is true**, or the else suite if all are false
- This example computes the absolute value of a difference, i.e., the distance between the numbers  $x$  and  $y$
- Python has the built-in abs function for this purpose, so this is just a pedagogical example.

```
1 if x > y:
2     absval = x - y
3 elif y > x:
4     absval = y - x
5 else:
6     absval = 0
```

- The Python for statement:

```
1 for target in sequence:  
2     [block of code]
```

- The for loop is a **sequence iteration that assign items in sequence to target one at a time** and runs the block of code for each item
- Unless the loop is terminated early with the `break` statement, the block of code is run as many times as there are items in the sequence

```
1 names = ['Peter', 'John', 'Mary', 'Helen', 'Tom', 'Nicholas']
2 for name in names:
3     print(name)
4 for x in [0,1,2,3,4,5,6,7,8,9,10]:
5     print(x)
6 for x in range(11):
7     print(x)
```

- There is also an optional `else` clause to the `for` loop, which is executed when the loop terminates through exhaustion of the sequence, but not when the loop is terminated by a `break` statement

- Dictionaries are often accessed and manipulated using the the `for` statement

```
1 age = {'Tim': 29, 'Jim': 31, 'Pam': 27, 'Sam': 35}
2 age.keys()
3 age.values()
4 for person in age:
5     print(person, age[person])
6 for person in age.keys():          # another way to achieve the same
7     print(person, age[person])
8 for person in sorted(age.keys()):
9     print(person, age[person])
10 for person in sorted(age.keys(), reverse=True):
11     print(person, age[person])
```

# Python for Statement & List Comprehensions

- A common operation is to take an existing list, apply some operation to all of the items of the list, and create a new list containing the results
- In Python there is an operator for this task known as a **list comprehension**
- Consider the following approach to computing squares of a list of numbers:

```
1 numbers = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
2 squares = []
3 for n in numbers:
4     squares.append(n*n)
```

- This can be easily implemented as a list comprehension:

```
1 numbers = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
2 squares = [n*n for n in numbers]
```

- If we just care about the squares, we can do the following:

```
1 squares = [n*n for n in range(0,11)]
```

- The general syntax is as follows:

```
1 [expression for item_1 in iterable_1 if condition_1
2     for item_2 in iterable_2 if condition_2
3     ...
4     for item_N in iterable_N if condition_N]
```



# Python for Statement & Handling Files

- We commonly want to read data from a file or write data to a file
- The syntax for reading a file line-by-line:

```
1 for line in open('input.txt'):
2     line = line.rstrip()
3     line = line.split(',')

```

- Or more economically:

```
1 for line in open('input.txt'):
2     line = line.rstrip().split(',')

```

- Why won't the following work?

```
1 for line in open("input.txt"):
2     line = line.split(',').rstrip()      # this will not work (!)

```

- Writing a file line-by-line:

```
1 F = open('output.txt', 'w')
2 F.write('Hello there! \n')      # need the newline character here
3 F.close()

```

- The default setting is that we are opening a file for reading, but this can also be made explicit:

```
1 F.open('input.txt', 'r')

```

- The Python `while` is used for repeated execution as long as an expression is true:

```
1 while expression:
2     [block of code]
3 else:
4     [block of code]
```

- The `while` loop repeatedly tests the expression
- If the expression is true, it executes the first block of code
- If the expression is false, it executes the second block of code, if present, and terminates the loop
- Use of `for` and `while` depends on whether looping over a known sequence or whether looping while some condition is true

# Python Functions

- Statements specify what the program should do
- End of line terminates a statement (no semicolons etc.)
- We can for example read input from the user and then print it out:

```
1 name = raw_input("Enter your name: ")
2 print("Hello " + name + "!")
```

- Backslash can be used to continue a (long) statement to the next line:

```
1 print("This is a very long line so it is very nice to be able to \
2     continue entering it on the next line.")
```

- Functions are devices for grouping statements so that they can be easily run more than once in a program
- Functions maximize code reuse and minimize code redundancy
- Functions enable dividing larger tasks into smaller chunks (procedural decomposition)
- Functions are written using the `def` statement
- The `return` statement sends the result object back to the caller

```
1 def add(a,b):  
2     result = a + b  
3     return result  
4 ...  
5 my_sum = add(2,3)
```

- All names assigned in a function are local to that function and exist only while the function runs (unless using the `global` statement)
- Arguments are matched by position
- Use **tuples** to return multiple values from a function:

```
1 def add_and_subtract(a,b):  
2     sm = a + b  
3     df = a - b  
4     return (sm, df)          # parenthesis are optional here  
5 ...  
6 my_sum, my_diff = add_and_subtract(5, 18)
```

- Functions do not exist until Python reaches and runs the `def` statement
- A function is not executed until the given function is called using the `function()` syntax
- The `def` statement creates an object and assigns it to a name

```
1 type(add)
```

```
<type 'function'>
```

```
1 newadd = add
2 type(newadd)
```

```
<type 'function'>
```

```
1 newadd(4,5)
```

- Arguments are passed by assigning objects to local names (object reference)
  - Mutable arguments are passed “by pointer” such that the original object gets modified
  - Immutable arguments are passed “by value” because it’s not possible to modify an immutable object so that a new object is created

```
1 def modify(mylist):  
2     mylist[0] *= 10  
3     ...  
4 L = [1,3,5,7,9]  
5 modify(L)  
6 print(L)
```

[10, 3, 5, 7, 9]



- Let's look at a slightly more complex example
- This function returns the intersection of two sequences

```
1 def intersect(seq1, seq2):  
2     res = []  
3     for x in seq1:  
4         if x in seq2:  
5             res.append(x)  
6     return res  
7 ...  
8 A = [1,3,5,7,9]  
9 B = [2,3,4,5,6]  
10 C = intersect(A,B)
```

- Remember polymorphism? The meaning of an operation depends on the type of the objects being operated upon
- For integers, the + operator means addition
- For strings, the + operator means concatenation

```
1 def add(a,b):  
2     return a + b  
3 ...  
4 add(12,23)  
5 add('Peter', 'John')
```

# Example: Password Generator

- Let's generate a simple program to generate passwords
- Would like to be able to specify the character set and password length
- `random.sample` is used for random sampling **without replacement**

```
1 import random
2
3 def password(length):
4     characters = "abcdefghijklmnopqrstuvwxyz"
5     pw = str()
6     for i in range(length):
7         pw = pw + random.choice(characters)
8     return pw
```

## Understanding Common Errors

- Not reading and/or understanding common error messages
- Not understanding basic built-in types (e.g., dictionaries have no internal ordering)
- Trying to do an operation that is not supported by the object
- Accessing the object in a wrong way (e.g., key vs. index)

- Trying to modify immutable objects
- Not understanding nested objects
- Not knowing the type of an object
- Trying to operate on two objects of different type (e.g., trying to concatenate a string and a number)