Python Crash-Course

Basics of Python Syntax

- No declared types
  - (x = 5 vs int x = 5)
- : Indentation vs “{ .. }“
- 0-based indexing
- Verbal commands
  - “and” rather than “&&”
- Object-oriented
- Logicals: True/False
- x**2 instead of x^2
- Casting ➜ int(), str()

```
for d in range(10,1000):
    num = Decimal(1) / Decimal(d)
    while num < 0.1:
        num = num * 10

    for patLength in range(1, decis/2):
        numStr = str(num*10**patLength)
        parts = numStr.split(".")
        if len(parts) > 1 and parts[1].startswith(parts[0]):
            if patLength > maxPattern:
                maxPattern = patLength
                maxD = d
            break
```

Conditionals and If-statements

- Only execute a portion of code if a certain condition is met
  - Pretty similar to other languages.

```
r = random.randint(0,101)
examGrade = "You got a "
if r > 90:
    examGrade += "A!"
elif r > 80:
    examGrade += "B!"
elif r > 70:
    examGrade += "C."
else:
    examGrade += "don't ask..."
```

- Types of conditionals
  - >,>=, <,<=, ==
- Combining multiple statements
  - and/or
- Negation
  - not

For Loops

- Runs through values in some sequence (iterables)
  - More like a Java “for each” loop (or MATLAB for-loop) rather than the standard Java for loop
  - Used for repeating lines of code on a series of inputs
- range(a,b) ➜ sequence from a to b, not including b: [a, a + 1, ..., b - 1]
  - Used for traditional for-loop behavior

```
sum = 0
for i in range(0,10):
    sum += i
# sum = 0 + 1 + 2 + ... + 9
```

- If iterating over a list of elements that have multiple parts, can extract several parts at once;

```
l = list([[1,2], [3,4], [5,6]])
for x,y in l:
    print "(" + str(x) + ", " + str(y) + ")"
```

```
(1, 2)
(3, 4)
(5, 6)
```

More Examples

- Enumerate pairs each element of the iterable with its index!

```
fibNums = list([0,1,1,2,3,5,8,13,21,34])
for index, num in enumerate(fibNums):
    print str(num) + " is at index " + str(index)
# enumerate pairs each element of the iterable with an index
# i.e. the order its read in, starting at 0.
```

```
0 is at index 0
1 is at index 1
1 is at index 2
2 is at index 3
3 is at index 4
5 is at index 5
s at index 6
is at index 7
is at index 8
is at index 9
```

- Zip(iterable1, iterable2, ...) ➜ creates a iterator that pairs the corresponding elements of each iterable together in tuples

```
l1 = [1,2,3,4,5]
l2 = [6,7,8,9,10]
l3 = [11,12,13,14,15]

print zip(l1,l2,l3)
```

```
[(1, 6, 11), (2, 7, 12), (3, 8, 13), (4, 9, 14), (5, 10, 15)]
```

While Loop

- Continue to do some operation while a condition is true
  - Helpful when we might not know when to stop

```
while not feelingSick:
    cookiesEaten += 1
    #random int 1 to 100
    r = random.randint(1,101)
    if r < cookiesEaten:
        feelingSick = True
print "I think " + str(cookiesEaten) + " was one too many..."
```

```
I think 16 was one too many...
```

Practice with For/While Loops

1) Find the cumulative sum of the first 1000 natural numbers.

2) Approximate log2 (10^6)

3) Given a list of positive integers, find the sum of the integers that match their index
    a) E.g. [0, 5 ,2, 2, 4,7, 7, 8, 10, 10, 4, 11, 2, 12, 14]

Iterables/Data Structures in Python

- Generic type of data structure that has elements which can be looped over
- Examples
  - Lists
  - Tuples (immutable lists)
  - Sets
  - Dictionaries (like a Java Map)
- Shared Methods
  - len(iterable) ➜ number of elements
  - str(iterable) ➜ string representation of the iterable
  - x in iterable ➜ returns true if x is in the given iterable, false otherwise
  - del s[index] ➜ removes value at a given index (lists/tuples only)

Lists

- Declaration: list([1,2,3]) or [1,2,3,4]
- Properties
  - Indexed (have a set order); list[index]
  - Mutable: list[index] = 6;
- Multiplication by integers
  - Expands list: [1,2,3] * 3 = [1,2,3,1,2,3,1,2,3]
- Comparable
  - [1,2,3] > [1,1,6] ➜ true (compares left to right; if equal, moves on; stops after finding difference)
- Helpful methods
  - list.remove(element) ➜ removes given element
  - list.append(element) ➜ adds element to end of list
  - list.insert(index, element) ➜ inserts element at given index (pushes elements out of way)
  - list.index(element) ➜ first index of element in list

Tuples

- Declaration: tuple([1,2,3]) or (1,2,3) or (50,)
- Immutable (cannot change the elements)
- Indexed (have a set order); x[index]
- Multiplication by Integer (same as list)
- Hashable
  - Can be used in non-ordered data structures, like a set or as a dictionary key (essentially it has some value associated to it that doesn't change, so the dict knows where to put it)
- Comparable
  - Compare same as lists

Handy Tricks

- Strings are iterable (lists)!
  - Can manipulate as other iterables (enumerate, for i in s, etc)
  - String manipulations are POWERFUL
- Slicing
  - Selects a range of values from an indexed (list/tuple) iterable
  - X[start:stop:step]; **stop is EXCLUSIVE**
    - If you leave one of these empty, by default ➔ start = 0; stop = end; step = 1
  - Negative indexes/increments
    - S[-1] returns last element
    - S[-3::] returns last 3 elements
    - S[::-1] returns s in reverse

```
s = "hello world"
reverse = s[::-1]
print reverse
```

```
dlrow olleh
```

```
#counts digits in number
num = 13747845892987147129847921010 9
digits = [0]*10
for d in str(num):
    digits[int(d)] += 1

print digits
```

```
[2, 5, 3, 1, 4, 1, 0, 5, 4, 5]
```

Practice problems with Lists/Tuples!

1. How would I extract all the elements of a list/tuple EXCEPT the last 3?

2. How would I extract all the elements of a list/tuple at odd-indexes?

3. I want to create the number 123456789987654321123456789987654321 quickly and efficiently, without typing it out. How would I do so?
    a. Hint: range(start, stop) will create a list of the integers between start and stop (not including stop)

Sets

- Declaration: set([ ]) or {a, b, c, d}
- Cannot store duplicate elements
- Helpful methods
  - s.isdisjoint(other) ➔ true if s and other have no elements in common; else false
  - s.add(element) and s.remove(element)

`s.issubset(t)`	`s <= t`	test whether every element in *s* is in *t*
`s.issuperset(t)`	`s >= t`	test whether every element in *t* is in *s*
`s.union(t)`	`s | t`	new set with elements from both *s* and *t*
`s.intersection(t)`	`s & t`	new set with elements common to *s* and *t*
`s.difference(t)`	`s - t`	new set with elements in *s* but not in *t*
`s.symmetric_difference(t)`	`s ^ t`	new set with elements in either *s* or *t* but not both
`s.copy()`		new set with a shallow copy of *s*

Dictionary

- Declaration: dict([(key, value), (key2, value2)]) or {key: value, key2:value2, ...}
  - dict(zip(iter1, iter2, iter3, ...)) ➜ discuss zip later
- dict.keys() ➜ returns a set of all keys in dictionary
- Retrieving values: d[key] ➜ value
- Modifying/adding values: d[key] = newValue
- Deleting keys: del d[key]

Practice with Dictionaries/Sets!

1. Given the dictionary key:value ➜ integer:integer constructed below, find the cumulative sum of the values associated to even keys
    a. Dictionary = dict(zip(range(1,100), range(251,350)))

2. Given two very large, repetitive numbers, determine what digits the two do not share.

Functions

- Define a set of operations for repeated use
- Default values
  - Unnecessary arguments
    - Listed using [parameter] in Python Documentation
- Returns vs reference semantics

```
# given a list, constructs random collection of its elements
# of the given size.
def randCombinations(elements, size, withReplace = False):
    combo = set([])
    # avoids damaging given data
    nums = elements[:]
    for i in range(0,size):
        r = random.randint(0,len(nums)-1)
        nextNum = nums[r]
        combo.add(nextNum)
        if not withReplace:
            nums.remove(nextNum)
    return combo

print randCombinations(list([1,2,3,4,5,6,7,8,9]), 3)
```

Lambda Operator

- Anonymous / In-line functions
- Convenient for creating short functions that are only used locally.

```
func = lambda input1, input2: input1 * math.exp(input2)
```

- filter(func, iterable)
  - Returns iterable with only elements that func(element) evaluates to true

```
evenNumbers = filter(lambda x: x%2 == 0, range(1,25))
print evenNumbers
[2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24]
```

More Operations!

- map(func, iterable)
  - Returns new iterable where func has been applied to each element of old iterable

```
squareNumbers = map(lambda x: x**2, range(1,11))
print squareNumbers
```

```
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

- reduce(func(x,y), iterable, [, initializer])
  - Takes first element = x; if there are elements left, applies func(x,y) with a remaining element = y. Let Xnew = f(x,y). If there are elements left, choose one = Ynew. Let func(Xnew, Ynew) = Xnew2..
  - Initializer; uses this as a first element if provided

```
cumulativeProd = reduce(lambda x,y: x*y, [1,2,3,4,5]) # = 120
cumulativeProd2 = reduce(lambda x,y: x*y, [1,1,1,1,1], 2)  # = 2
```

Practice with Functions/Lambda!

1) Create a function that determines whether a number is prime or not

2) In one-line, create a list of all the prime numbers under 100,000.

3) Which digit is most common amongst these numbers?

Generators (Briefly)

- Function
  - give me parameters, I spit out an object (i.e. a list)
  - **return**
- Generator
  - give me parameters, I'll spit out one element at a time every time you call me
  - SPACE-EFFICIENT
  - **yield**

Examples!

```
def fibNumber():
    last = 0;
    current = 1;
    while True:
        next = last + current
        yield next
        last = current
        current = next
```

```
f = fibNumber()
print f.next()
print f.next()
print f.next()
print f.next()
print f.next()
```

```
1
2
3
5
8
```

```
#for loops essentially just keep calling .next()
for n in fibNumber():
    print n
    time.sleep(1)
```

Classes

- Allows you to make an object that encapsulates a specific state and set of behaviors
- Useful for helping others use a tool that they might not have the background to use
- E.g. Chromestat
  - No idea how it works but maybe I need to be able to use it
  - Maybe, all I need to do is take measurements and export data; I don’t need to know how it works to understand those, so if someone can write code to extract them for me, problem solved!

```
import numpy as np

class PointCollection:
    def __init__(self, tolerance, start, win):
        self.items = [start]
        self.size = 0
        self.tol = tolerance
        self.edges = {}
        self.lastPoint = start
        self.win = win
        Circle(startPoint, 4).draw(win)

    def add(self, other):
        print(self.items)
        self.items = self.items + [other]
        self.size += 1

    def contains(self, point):
        for k in range(self.size):
            if np.absolute(self.items[k].x - point.x) < self.tol and np.absolute(self.items[k].y - point.y) < self.tol
                print("No new connection made")
                return True

        return False

    def drawSelf(self, newPt):
        Circle(newPt, 4).draw(self.win)
        newPt.draw(self.win)
        line = Line(newPt, self.lastPoint)
        line.draw(self.win)
        self.lastPoint = newPt
```

Example Point Collection

Recursion

- Having a method call itself, until some sort of break condition is met
- Problems that can be broken down into nested cases
  - If I knew func(N-1), I could certainly calculate func(N)...

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n*factorial(n-1)

for n in range(0, 7):
    print str(n) + "! = " + str(factorial(n))
```

```
0! = 1
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
```

Recursion Practice!

The following iterative sequence is defined for the set of positive integers:

$n \rightarrow n/2$ (n is even)
$n \rightarrow 3n + 1$ (n is odd)

Using the rule above and starting with 13, we generate the following sequence:

$$13 \rightarrow 40 \rightarrow 20 \rightarrow 10 \rightarrow 5 \rightarrow 16 \rightarrow 8 \rightarrow 4 \rightarrow 2 \rightarrow 1$$

It can be seen that this sequence (starting at 13 and finishing at 1) contains 10 terms. Although it has not been proved yet (Collatz Problem), it is thought that all starting numbers finish at 1.

Which starting number, under one million, produces the longest chain?

NOTE: Once the chain starts the terms are allowed to go above one million.

File-Parsing

- Lots of useful information online comes in files
- Save file to working directory (easiest way)
  - Alternatively, you'll have to indicate some kind of path; full or from working directory

```
with(open("/Users/zackmcnulty/Desktop/iGEM/journals/SSH_stuff/test.txt")) as inputFile:
    for line in inputFile:
        print line
```

```
this is a test.

It is not a long test.

But a test

Nonetheless
```

- With open … as variableName:
  - Creates an iterable object where each entry is a line of the file
  - Can specify file location as path from working directory, or absolute path

File-Parsing Practice!

1) Download the file find_the_i.txt on the simulations gitHub page and find what line the i is on (it’s a capital I)!

2) Using the online documentation for open(), or any other such resource, determine how to write the string “I did it!” to the file success.txt.

Useful Modules/Libraries

- Library: A collection of cool functions you didn't have to write the code for!
  - One for practically everything
- Using a library:
  - Step 1: Install it on your device:

```
pip install BeautifulSoup
```

  - Step 2: import it to your python project (rename for convenience)

```
import numpy as np
import matplotlib.pylab as plt
```

  - Step 3 (not needed for Spyder through Anaconda): Implement it in the Python Interpreter you are using
  - Step 4: Call a method ➜ plt.plot()

Numpy

- Scientific Computing: Arrays/Matrices and more!
  - Optimized in C++ to be fast!
  - Matrix operations, element-by-element operations, etc.
  - Implemented in many other libraries; a foundation of python scientific computing
- Element-wise operations by default
- https://docs.scipy.org/doc/numpy-1.13.0/user/basics.broadcasting.html#module-numpy.doc.broadcasting

Numpy n-dimensional Arrays

- Features/Fields
  - a.ndim ➔ number dimensions in array
  - a.shape ➔ tuple listing size of the array in each dimension
  - a.size ➔ total number of elements
  - a.dtype ➔ data type being stored in array
- Generating Arrays
  - np.arange(start = 0, stop, step = 1) ➔ generates 1D vector [start, stop) with increments step
    - array.reshape(size dim1, size dim2, ...) ➔ changes shape of an array to new dimensions
  - np.array([(1,2,3), (4,5,6), (7,8,9)]) ➔ each tuple represents a row of matrix
  - np.zeros((size dim1, size dim2, ...)) ➔ array of given dimensions filled with zeros
  - Np.ones ((size dim1, size dim2, ...))
  - np.fromfunction(func(x,y, ...), (size dim1, size dim2, ...))
    - Number of parameters in function matches number of dimensions given (remember zero based indexing)

More

- Functions and Constants
  - np.pi, np.exp(), np.sin(), etc.
- Operations
  - array.sum(), array1.dot(array2), array.max(), ... pretty much any operation you could think of
- Accessing Elements in Array
  - Same as accessing elements in a list; just more dimensions

Practice with Numpy!

1. Create a 6 x 6 array, where all elements on the outside edge of the matrix are 1’s, and all elements on the inside are 0’s

2. Given the 3 x 2 matrix below of cartesian (x,y) coordinates, convert the coordinates to polar
   a. a = np.transpose(np.array([(np.sqrt(2)/2, np.sqrt(3)/2, 0.5), (np.sqrt(2)/2, 0.5, np.sqrt(3)/2)]))
   b. If you forgot the conversion: r = sqrt(x^2 + y^2), tan theta = y/x

Itertools

- Creating iterators for efficient looping!
  https://docs.python.org/2/library/itertools.html

Iterator	Arguments	Results
`product()`	p, q, … [repeat=1]	cartesian product, equivalent to a nested for-loop
`permutations()`	p[, r]	r-length tuples, all possible orderings, no repeated elements
`combinations()`	p, r	r-length tuples, in sorted order, no repeated elements
`combinations_with_replacement()`	p, r	r-length tuples, in sorted order, with repeated elements

`chain()`	p, q, …	p0, p1, … plast, q0, q1, …	`chain('ABC', 'DEF') --> A B C D E F`
`compress()`	data, selectors	(d[0] if s[0]), (d[1] if s[1]), …	`compress('ABCDEF', [1,0,1,0,1,1]) --> A C E F`
`dropwhile()`	pred, seq	seq[n], seq[n+1], starting when pred fails	`dropwhile(lambda x: x<5, [1,4,6,4,1]) --> 6 4 1`
`groupby()`	iterable[, keyfunc]	sub-iterators grouped by value of keyfunc(v)	
`takewhile()`	pred, seq	seq[0], seq[1], until pred fails	`takewhile(lambda x: x<5, [1,4,6,4,1]) --> 1 4`

Examples

```
keys = []
groups = []

nums = range(1,10)
isEven = lambda x: x %2 == 0
# pairs into groups of elements in num
# that evaluate to the same value under
# the given function. Thus, if you want
# to group entire list, sort it based
# on that same function first.
nums = sorted(nums, key=isEven)
for k,g in itl.groupby(nums, isEven):
    keys.append(k)
    groups.append(list(g))

print zip(keys, groups)
```

```
[(False, [1, 3, 5, 7, 9]), (True, [2, 4, 6, 8])]
```

```
# probability two people in a room of 3 share a birthday
days = range(0,365)
combos = list(itl.product(days, repeat = 3))
sameBDay = filter(lambda x: x[0] == x[1], combos)
print len(sameBDay)*1.0/len(combos)
```

Practice with Itertools!

1. Given a random list of numbers, write code to find the longest sequence of consecutive prime numbers.
    a. L1 = [1,2,3,4,5,7,11,13,17,21,32,34,45,54,54,65,12,32]

2. What is the probability that in a game of blackjack you get 21 on your first 2 cards?

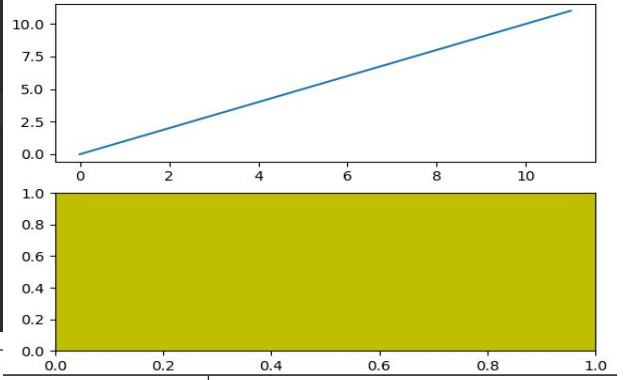
Matplotlib

- MATLAB-esque plotting
- https://matplotlib.org/users/index.html
- Useful Commands
  - hist(data) ➔ given a bunch of data points, plots a histogram
  - scatterplot(x,y)
  - plot(x,y)
    - subplot(rows, columns, plotNumber)
    - semilogx(), semilogy(), loglog()
- **plt.show() ➔ not all plots show up automatically!!!**
- LOTS of additional parameters to alter style of graph

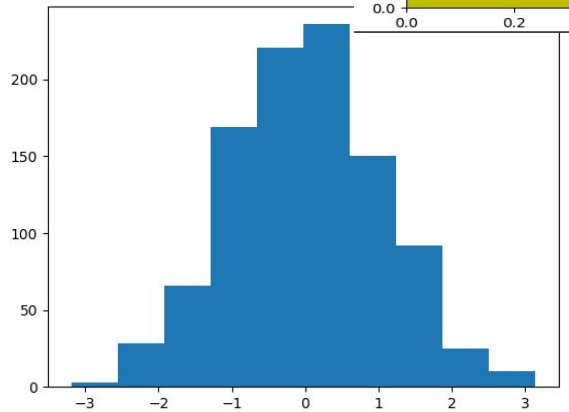
Examples Matplotlib!

```
import matplotlib.pyplot as plt
# now create a subplot which represents the top plot of a grid
# with 2 rows and 1 column. Since this subplot will overlap the
# first, the plot (and its axes) previously created, will be removed
plt.subplot(2,1,1)
plt.plot(range(12))
plt.subplot(2,1,2, facecolor='y') # creates 2nd subplot with yellow background
plt.show()

time.sleep(100)
```




```
ys = np.random.normal(0,1,1000)
plt.hist(ys)
plt.show()
```



Practice Problem

Pete and Colin are playing a game involving dice. Pete has four 4-sided dice with the numbers 1-4 and Colin has three 6-sided dice labeled 1-6 that they roll at the same time.

1. Make 2 probability distributions (histograms) for the possible total sums that each may get after a single roll and plot them side-by-side.
2. What is the chance that Pete wins any given game (don’t think statistics)?

Other Useful Libraries

SymPy - Symbolic (numerical) Mathematics

Pandas - data frames, tables, higher-order structures & data analysis tools

Biopython - Computational Molecular Biology Tools

Questions? More Problems?